CHAPTER 37

DEFECT AND FAULT TOLERANCE

André DeHon
Department of Electrical and Systems Engineering
University of Pennsylvania

As device size F continues to shrink, it approaches the scale of individual atoms and molecules. In 2007, 65-nm integrated circuits are in volume production for processors and field-programmable gate arrays (FPGAs). With atom spacing in a silicon lattice around 0.5 nm, F = 65-nm drawn features are a little more than 100 atoms wide. Key features, such as gate lengths, are effectively half or a third this size. Continued geometric scaling (e.g., reducing the feature size by a factor of 2 every six years) will take us to the realm where feature sizes are measured in single-digit atoms sometime in the next couple of decades.

Very small feature sizes will have several effects on integrated circuits, including:

- *Increased defect rates:* Smaller devices and wires made of fewer atoms and bonds are less likely to be "good enough" to function properly.
- *Increased device variation:* When dimensions are a few atoms wide, the addition, absence, or exact position of each atom has a significant affect on device parameters.
- *Increased change in device parameters during operational lifetime:* With only a few atoms making up the width of wires or devices, small changes have large impacts on performance, and the likelihood of a complete failure grows. The fragility of small devices reduces traditional opportunities to overstress them as a means of forcing weak devices to fail before the component is integrated into an end system. This means many weak devices will only turn into defects during operation.
- *Increased single die capacity:* Smaller devices allow integration of more devices per die. Thus, not only do we have devices that are more likely to fail, but there also are more of them, meaning more chances that some device on the die will fail.
- *Increased susceptibility to transient upsets:* Smaller nodes use less charge to hold state or configuration data, making them more susceptible to upset by noise, including ionizing particles, thermal noise, and shot noise. Coupled with the greater capacity, which means more nodes that can be upset, dies will have significantly increased upset rates.

Accommodating and exploiting these effects will demand an increasing role for postfabrication configurable architectures. Nonetheless, some usage paradigms

will need to shift to fully exploit the potential benefits of reconfigurable architectures at the atomic scale.

This chapter reviews defect tolerance approaches and points out how the configurability available in reconfigurable architectures is a key tool for coping with defects. It also touches briefly on lifetime and transient faults and their impact on configurable designs.

37.1 DEFECTS AND FAULTS

A *defect* is a persistent error in a component. Because defects are persistent, we can test for defect occurrences and record their locations. We contrast defects with transient faults that may produce the wrong value on one or a few cycles but do not continue to corrupt calculations. For the sake of simple discussion here, we classify any persistent problem that causes the circuitry to work incorrectly for some inputs and environments as defects. Defects are often modeled as stuck-at-1, stuck-at-0, or shorted nodes. They can also be nodes that are excessively slow, such that they compute correctly but not in a timely fashion, or excessively leaky, such that they do not hold their value properly. A large number of physical effects and causes may lead to these manifestations, including broken wires, shorts or bridging between nodes that should be distinct, excessive or inadequate doping in a device, poor contacts between materials or features, or excessive variation in device size.

A *transient fault* is a temporary error in a circuit result. Transient faults can occur at random times. A transient fault may cause a gate output or node to take on the incorrect value on some cycle of operation. Examples of transient faults include ionizing particles (e.g., α-particles), thermal noise, and shot noise.

37.2 DEFECT TOLERANCE

37.2.1 Basic Idea

An FPGA or reconfigurable array is a set of identical (programmable) bit-processing operators with postfabrication configurable interconnect. When a device failure renders a bitop or an interconnect segment unusable, we can configure the computation to avoid the failing bitop or segment (see Figure 37.1). If the bitop is part of a larger SIMD word (Chapter 36, Section 36.3.2) or other structure that does not allow its independent use, we may be forced to avoid the entire structure. In any case, as long as all the resources on the reconfigurable array are not being used, we can substitute good resources for the bad ones. As defect rates increase, this suggests a need to strategically reserve spare resources on the die so that we can guarantee there are enough good resources to compensate for the unusable elements.

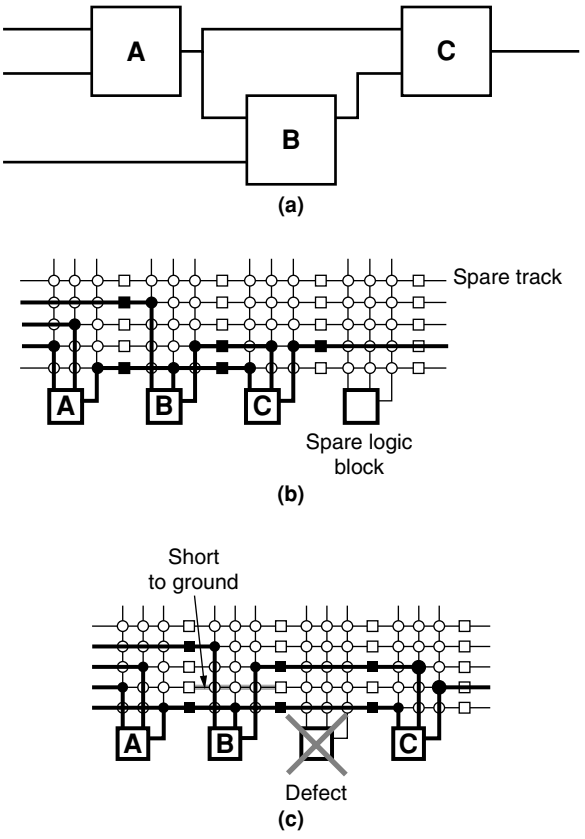

FIGURE 37.1 ■ Configuring computation to avoid defective elements in a reconfigurable array: (a) logical computation graph, (b) mapping to a defect-free array with spare, and (c) mapping to an array with defects.

This basic strategy of (1) provisioning spare resources, (2) identifying and avoiding bad resources, and (3) substituting spare resources for bad resources is well developed for data storage. DRAM and SRAM dies include spare rows and columns and substitute the spare rows and/or columns for defective rows and columns (e.g., see [1, 2]). Magnetic data storage (e.g., hard disk) routinely has bad sectors; the operating system (OS) maps the bad sectors and takes care not to allocate data to those sectors. These two forms of storage actually illustrate two models for dealing with defects:

1. *Perfect component:* In the perfect component model, the component has to look perfect; that is, we require every address visible to the user to perform correctly. The spare resources are added beyond those required to deliver the promised memory capacity and are substituted out behind the scenes so that users never see that there are defective elements in the component. DRAM and SRAM components are the traditional example of the perfect component model.

2. *Defect map:* The defect map model allows elements to be bad. We expose these defects to higher levels of software, typically the OS, which is responsible for tracking where the defects occur and avoiding them. Magnetic disks are a familiar example of the defect map model—we permit sectors to be bad and format the disk to avoid them.

37.2.2 Substitutable Resources

Some defects will be catastrophic for the entire component. While a reconfigurable array is composed largely of repeated copies of identical instances, the device infrastructure is typically unique; defects in this infrastructure may not be repairable by substitution. Common infrastructures include power and ground distribution, clocking, and configuration loading or instruction distribution. It is useful to separate the resources in the component into *nonrepairable* and *repairable* resources. Then we can quantify the fraction of resources that are nonrepairable.

We can minimize the impact of nonrepairable resources either by reducing the fraction of things that cannot be repaired or by increasing the reliability of the constituent devices in the nonrepairable structures. Many of the infrastructure items, such as power and ground networks, are built with larger devices, wires, and feature sizes. As such, they are less susceptible to the failures that impact small features. Memory components (e.g., DRAMs) also have distinct repairable and nonrepairable components; they typically use coarser feature sizes for the nonrepairable infrastructure. Memory designs only use the smallest features for the dense memory array, where row and column sparing can be used to repair defects. In FPGAs, it may be reasonable to provide spares for some of the traditional infrastructure items to reduce the size of the nonrepairable region. For example, modern FPGAs already include multiple clock generators and configurable clock trees; as such, it becomes feasible to repair defective clock generators or portions of the clock tree by substitution. We simply need to guarantee that there are sufficient alternative resources to use instead of the defective elements.

For any design there will be a minimum *substitutable unit* that defines the granularity of substitution. For example, in a memory array we cannot substitute out individual RAM cells. Rather, with a technique like row sparing, the substitutable unit is an entire row. In the simplest sparing schemes, a defect anywhere within a substitutable unit may force the discard of the entire element. Consequently, the granularity of substitution can play a big role in the viable yield of a component (see the Perfect yield subsection that follows). Section 37.2.5 examines more sophisticated sparing schemes that relax this constraint.

37.2.3 Yield

This section reviews simple calculations for the yield of components and substitutable units. We assume uniform device defect rates and independent, random

failure (i.e., identical, independently distributed—iid). Using these simple models, we can illustrate the kinds of calculations involved and build intuition on the major trends.

Perfect yield

A component with no substitutable units will be nondefective only if all the devices in the unit are not defective. Similarly, in the simplest models each substitutable unit is nondefective only when all of its constituent devices are not defective. If we have a device defect probability P_d and if a unit contains N devices, the probability that the entire component or unit is nondefective is:

$$P_{defect-free}(N, P_d) = (1 - P_d)^N \tag{37.1}$$

We can expand this as a binomial:

$$P_{defect-free}(N, P_d) = \sum_i \left(\binom{N}{i} (-P_d)^i \right) = 1 - N \cdot P_d + \binom{N}{2} (P_d)^2 - \ldots \tag{37.2}$$

If $N \times P_d << 1$, then we observe that each successive power of P_d is much smaller than the previous term. We can approximate this yield as:

$$P_{defect-free}(N, P_d) \approx 1 - N \cdot P_d \tag{37.3}$$

This tells us we have a substitutable unit defect rate, P_{sd}, or a component defect rate, roughly equal to the product of the number of devices and the device defect rate:

$$P_{sd}(N, P_d) \approx N \cdot P_d \tag{37.4}$$

This simple equation indicates several things:

- For today's large chips with $N > 10^9$ devices, the defect rate P_d must be below 10^{-10} to expect 90 percent or greater chip yield.
- To maintain constant yield ($P_{defect-free}$) for a chip as N scales, we must continually decrease P_d at the same rate. For example, a 10× increase in device count, N, must be accompanied by a 10× decrease in per-device defect rate.
- As noted in this chapter's introduction, we expect the opposite effect for atomic-scale devices; smaller devices mean a higher likelihood of defects. This exacerbates the challenge of increasing device counts.
- At the same defect rate, P_d, a finer-grained substitutable unit (e.g., an individual LUT or bitop) will have a higher unit yield rate than a coarser-grained unit (e.g., a cluster of 10 LUTs, such as an Altera LAB (Section 1.5.1) or an SIMD collection of 32 bitops). Alternatively, if one reasons about defect rates of the substitutable units, a defect rate of $P_{sd} = 0.05$ for a coarse-grained block corresponds to a much lower device defect rate, P_d, than the same P_{sd} for a fine-grained substitutable unit.

- To keep substitutable unit yield rates at some high value, we must decrease unit size, N, as P_d increases. For example, if we design for a $P_{sd} = 10^{-4}$ and the device defect rate doubles, we need to cut the substitutable block size in half to achieve the same block yield; this suggests a trend toward fine-grained resource sparing as defect rates increase (e.g., see Fine-grained Pterm matching subsection of Section 37.2.5 and Section 38.6).

Yield with sparing

We can significantly increase overall yield by providing spares so that there is no need to demand that every substitutable unit be nondefective. Assume for now that all substitutable units are interchangeable. The probability that we will have exactly i nondefective substitutable units is:

$$P_{yield}(N, i) = \left(\binom{N}{i} (P_{sd})^i (1 - P_{sd})^{N-i} \right) \tag{37.5}$$

That is, there are $\binom{N}{i}$ ways to select i nondefective blocks from N total blocks, and the yield probability of each case is $(P_{sd})^i (1 - P_{or})^{N-i}$. An ensemble with at least M items is obtained whenever M or more items yield, so the ensemble yield is actually the cumulative distribution function, as follows:

$$P_{yield}(N, M) = \sum_{M \le i \le N} \left(\binom{N}{i} (P_{sd})^i (1 - P_{sd})^{N-i} \right) \tag{37.6}$$

As an example, consider an Island-style FPGA cluster (see Figure 37.2) composed of 10 LUTs (e.g., Altera LAB, Chapter 1). Assume that each LUT, along with its associated interconnect and configuration, is a substitutable unit and that the LUTs are interchangeable. Further, assume $P_{sd} = 10^{-4}$. The probability of yielding all 10 LUTs is:

$$P_{yield}(10, 10) = \left(10^{-4}\right)^{10} \left(1 - 10^{-4}\right)^{0} \approx 0.9990005 \tag{37.7}$$

Now, if we add two spare lookup tables, the probability of yielding at least 10 LUTs is:

$$\begin{aligned} P_{yield}(12, 10) &= \left(10^{-4}\right)^{12} \left(1 - 10^{-4}\right)^{0} + 12 \left(10^{-4}\right)^{11} \left(1 - 10^{-4}\right)^{1} \\ &\quad + \frac{12 \cdot 11}{2} \left(10^{-4}\right)^{10} \left(1 - 10^{-4}\right)^{2} \\ &= 0.99880065978 + 0.0011986806598 + 0.0000006593402969 \\ &\approx 0.9999999998 > 1 - 10^{-9} \end{aligned} \tag{37.8}$$

Without the spares, a component with only 1000 such clusters would be difficult to yield. With the spares, components with 1,000,000 such clusters yield more than 99.9 percent of the time.

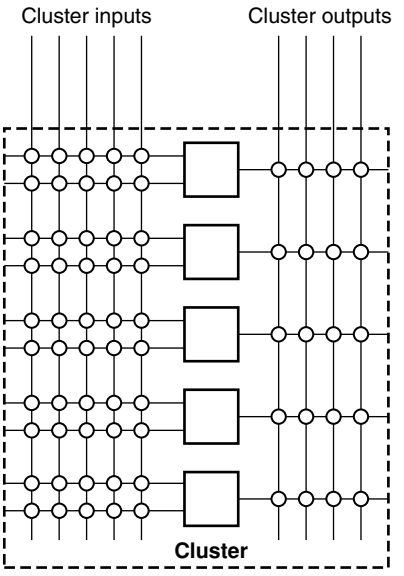

FIGURE 37.2 ■ An island-style FPGA cluster with five interchangeable 2-LUTs.

The assumption that all substitutable units are interchangeable is not directly applicable to logic blocks in an FPGA since their location strongly impacts the interconnections available to other logic block positions. Nonetheless, the sparing yield is illustrative of the trends even when considering interconnect requirements.

To minimize the required spares, it would be preferable to have fewer large pools of mostly interchangeable resources rather than many smaller pools of interchangeable resources. This results from Bernoulli's Law of Large Numbers (the Central Limit Theorem) effects [3, 4], where the variance of a sum of random variables decreases as the number of variables increases. For a more detailed development of the impact of the Law of Large Numbers on defect yield statistics and strategies see DeHon [5].

37.2.4 Defect Tolerance through Sparing

To exploit substitution, we need to locate the defects and then avoid them. Both testing (see next subsection) and avoidance could require considerable time for each individual device. This section reviews several design approaches, including approaches that exploit full mapping (see the Global sparing subsection) to minimize defect tolerance overhead, approaches that avoid any extra mapping (see the Perfect component model subsection), and approaches that require only minimal, local component-specific mapping (see the Local sparing subsection).

Testing

Traditional acceptance testing for FPGAs (e.g., [6]) attempts to validate that the FPGA is defect free. Locating the position of any defect is generally not

important if any chip with defects is discarded. Identifying the location of all defects is more difficult and potentially more time consuming. Recent work on group testing [7–9] has demonstrated that it is possible to identify most of the nondefective resources on a chip with N substitutable components in time proportional to $\sqrt{N}$.

In group testing, substitutable blocks are configured together and given a self-test computation to perform. If the group comes back with the correct result, this is evidence that everything in the group is good. Conversely, if the result is wrong, this is evidence that something in the group may be bad. By arranging multiple tests where substitutable blocks participate in different groups (e.g., one test set groups blocks around rows while another groups them along columns), it is possible to identify which substitutable units are causing the failures.

For example, if there is only one failure in each of two groupings, and the failing groups in each grouping contain a single, common unit, this is strong evidence that the common unit is defective while the rest of the substitutable units are good. As the failure rates increase such that multiple elements in each group fail in a grouping, it can be more challenging to precisely identify failing components with a small number of groupings. As a result, some group testing is conservative, marking some good components as potential defects; this is a trade-off that may be worthwhile to keep testing time down to a manageably low level as defect rates increase.

In both group testing and normal FPGA acceptance testing, array regularity and homogeneity make it possible to run tests in parallel for all substitutable units on the component. Consequently, testing time does not need to scale as the number of substitutable units, N. If the test infrastructure is reliable, group tests can run completely independently. However, if we rely on the configurable logic itself to manage tests and route results to the test manager, it may be necessary to validate portions of the array before continuing with later tests. In such cases, testing can be performed as a parallel wave from a core test manager, testing the entire two-dimensional device in time proportional to the square root of the number of substitutable units (e.g., [8]).

Global sparing

A defect map approach coupled with component-specific mapping imposes low overhead for defect tolerance. Given a complete map of the defects, we perform a component-specific design mapping to avoid the defects. Defective substitutable units are marked as bad, and scheduling, placement, and routing are performed to avoid these resources. An annealing placer (Chapter 14) can mark the physical location of the defective units as invalid or expensive and penalize any attempts to assign computations to them. Similarly, a router (Chapter 17) can mark defective wires and switches as "in use" or very costly so that they are avoided. The Teramac custom-computing machine tolerated a 10 percent defect rate in logic cells ($P_{sd_{logic}} = 0.10$) and a 3 percent defect rate in on-chip interconnect ($P_{sd_{interconnect}} = 0.03$) using group testing and component-specific mapping [7].

With place-and-route times sometimes running into hours or days, the component-specific mapping approach achieves low overhead for defect tolerance at the expense of longer mapping times. As introduced in Chapter 20, there are several techniques we could employ to reduce this mapping time, including:

- Tuning architectures to facilitate faster mapping by overprovisioning resources and using simple architectures that admit simple mapping; the Plasma chip—an FPGA-like component, which was the basis of the Teramac architecture—takes this approach and was highlighted in Chapter 20.
- Trading mapping quality in order to reduce mapping time.
- Using hardware to accelerate placement and routing (also illustrated in Sections 9.4.2 and 9.4.3).

Perfect component model

To avoid the cost of component-specific mapping, an alternate technique to use is the perfect component model (Section 37.2.1). Here, the goal is to use the defect map to preconfigure the allocation of spares so that the component looks to the user like a perfect component. Like row or column sparing in memory, entire rows or columns may be the substitutable units. Since reconfigurable arrays, unlike memories, have communication lines between blocks, row or column sparing is much more expensive to support than in memories. All interconnect lines must be longer, and consequently slower, to allow configuration to reach across defective rows or columns. The interconnect architecture must be designed such that this stretching across a defective row is possible, which can be difficult in interconnects with many short wires (see Figure 37.3).

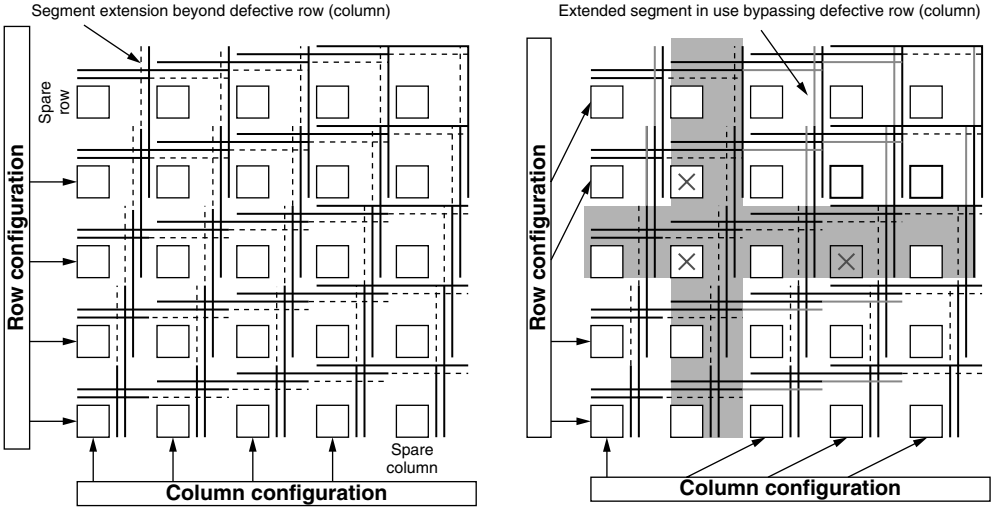

FIGURE 37.3 ■ Arrays designed to support row and column sparing.

A row of FPGA logic blocks is a much coarser substitutable unit than a memory row. FPGAs from Altera have used this kind of sparing to improve component yield [10, 11], including the Apex 20KE series.

Local sparing

With appropriate architecture or stylized design methodology, it is possible to avoid the need to fully remap the user design to accommodate the defect map. The idea here is to guarantee that it is possible to locally transform the design to avoid defects. For example, in cases where all the LUTs in a cluster are interchangeable, if we provision spares within each cluster as illustrated earlier in the Yield with sparing subsection of Section 37.2.3, it is simply a matter of locally reassigning the functions to LUTs to avoid the defective LUTs.

For regular arrays, Lach et al. [12] show how to support local interchange at a higher level without demanding that the LUTs exist in a locally interchangeable cluster. Consider a $k \times k$ tile in the regular array. Reserve s spares within each $k \times k$ tile so that we only populate $(k^2 - s)$ LUTs in each such region. We can now compute placements for the $(k^2 - s)$ LUTs for each of the possible combinations of s defects. In the simplest case, $s = 1$, we precalculate k^2 placements for each region (e.g., see Figure 37.4). Once we have a defect map, as long as each region has fewer than s errors, we simply assemble the entire configuration by selecting an appropriate configuration for each tile.

When a routing channel provides full crossbar connectivity, similarly, it may be possible to locally swap interconnect assignments. However, typical FPGA routing architectures do not use fully populated switching; as a result, interconnect sparing is not a local change. Yu and Lemieux [13, 14] show that FPGA switchboxes can be augmented to allow local sparing at the expense of 10 to 50 percent of area overhead. The key idea is to add flexibility to each switchbox that allows a route to shift one (or more) wire track(s) up or down; this allows routes to be locally redirected around broken tracks or switches and then restored to their normal track (see Figure 37.5).

To accommodate a particular defect rate and yield target, local interchange will require more spares than global mapping (see the Global sparing subsection). Consider any of the local strategies discussed in this section where we allocate one spare in each local interchange region (e.g., cluster, tile, or channel). If there are two defects in one such region, the component will not be repairable. However, the component may well have adequate spares; they are just assigned to different interchange regions. With the same number of resources, a global remapping would be able to accommodate the design. Consequently, to achieve the same yield rate as the global scheme, the local scheme always has to allocate more spares. This is another consequence of the Law of Large Numbers (see the Yield with sparing subsection):

> *The more locally we try to contain replacement, the higher variance we must accommodate, and the larger overhead we pay to guarantee adequate yield.*

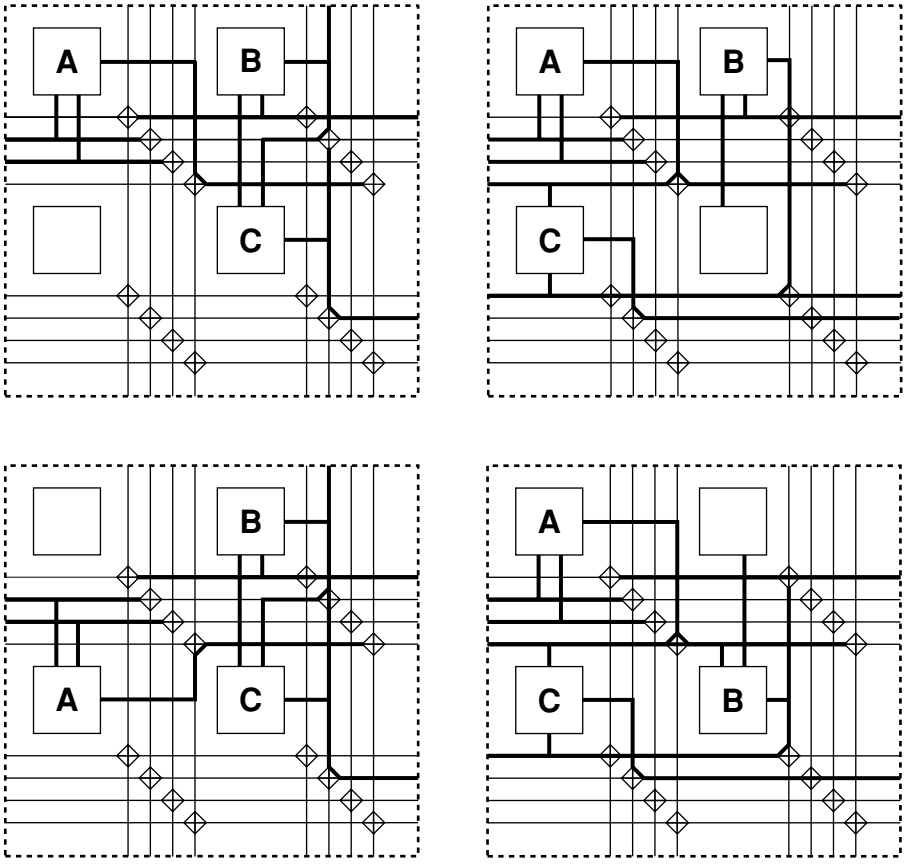

FIGURE 37.4 ■ Four placements of a three-gate subgraph on a 2×2 tile.

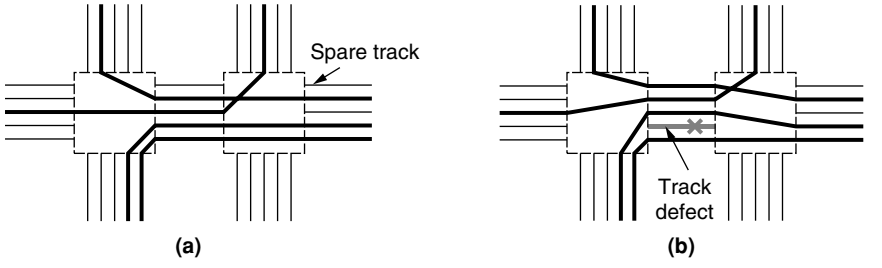

FIGURE 37.5 ■ Added switchbox flexibility allows local routing around interconnect defects: (a) defect free with spare and (b) configuration avoiding defective track.

37.2.5 Defect Tolerance with Matching

In the simple sparing case (Section 37.2.4), we test to see whether each substitutable unit is defect free. Substitutable units with defects are then avoided. This works well for low-defect rates such that P_{sd} remains low. However, it can also be highly conservative. In particular, not all capabilities of the substitutable unit are always needed. A configuration of the substitutable unit that avoids the particular defect may still work correctly. Examples where we may not need to use all the devices inside a substitutable unit include the following:

- A typical FPGA logic block, logic element, or slice includes an optional flip-flop and carry-chain logic. Many of the logic blocks in the user's design leave the flip-flop or carry chain unused. Consequently, these "defective" blocks may still be usable, just for a subset of the logical blocks in the user's design.
- When the substitutable unit is a collection of W_{simd} bitops, a defect in one of the bitops leaves the unit imperfect. However, the unit may work fine on smaller data. For example, maybe a $W_{simd} = 8$ substitutable unit has a defect in bit position 5. If the application requires some computations on $W_{app} = 4$ bit data elements, the defective 8-bit unit may still perform adequately to support 4 bitops.
- A product term (Pterm) in a programmable logic array (PLA) or programmable array logic (PAL) is typically a substitutable unit. Each Pterm can be configured to compute the AND of any of the inputs to the array (see Figure 37.6). However, all the Pterms configured in the array will never need to be connected to all the inputs. Consequently, defects that prevent a Pterm from connecting to a subset of the inputs may not inhibit it from being configured to implement some of the Pterms required to configure the user's logic.

Instead of discarding substitutable units with defects, we characterize their capabilities. Then, for each logical configuration of the substitutable unit

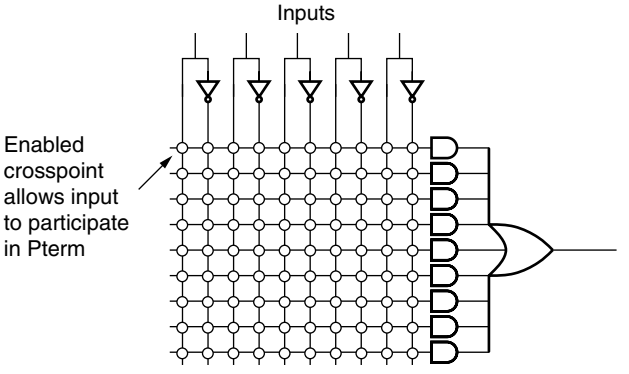

FIGURE 37.6 ■ A PAL OR-term with a collection of substitutable Pterm inputs.

demanded by the user's application, we can identify the set of (potentially defective) substitutable units capable of supporting the required configuration. Our mapping then needs to ensure that assignments of logical configurations to physical substitutable units obey the compatibility requirements.

Matching formulation

To support the use of partially defective units as substitutable elements, we can formulate the mapping between logical configurations and substitutable units as a bipartite matching problem. For simplicity and exposition, it is assumed that all the substitutable units are interchangeable. This is likely to be an accurate assumption for LUTs in a cluster or Pterms in a PAL or PLA, but it is not an accurate assumption for clusters in a two-dimensional FPGA routing array. Nonetheless, this assumption allows precise formulation of the simplest version of the problem.

We start by creating two sets of nodes. One set, $R = \{r_0, r_1, r_2 \ldots\}$, represents the physical substitutable resources. The second set, $L = \{l_0, l_1, l_2 \ldots\}$, represents the logic computations from the user's design that must be mapped to these substitutable units. We add a link (l_i, r_j) if-and-only-if logical configuration l_i can be supported by physical resource r_j. This results in a bipartite graph, with L being one side of the graph and R being the other. What we want to find is a *complete matching* between nodes in L and nodes in R—that is, we want every $l_i \in L$ to be matched with exactly one node $r_j \in R$, and every node $r_j \in R$ to be matched with at most one node $l_i \in L$.

We can optimally compute the *maximal matching* between L and R in polynomial time using the Ford–Fulkerson maximum flow algorithm [15] with time complexity $O(|V| \cdot |E|)$ or a Hopcroft–Karp algorithm [16] with time complexity $O\left(\sqrt{|V|} \cdot |E|\right)$. In the graph, $|V| = |L| + |R|$ and $|E| = O(|L| \cdot |R|)$. Since there must be at least as many resources as logical configurations, $|L| \leq |R|$, the Hopcroft–Karp algorithm is thus $O\left(|R|^{2.5}\right)$; for local sparing schemes, $|R|$ might be reasonably in the 10 to 100 range, meaning that the matching problem is neither large nor growing with array size. If the maximal matching fails to be a complete matching (i.e., assign each l_i to a unique match in r_i), we know that it is not possible to support the design on a particular set of defective resources.

Fine-grained Pterm matching

Naeimi and DeHon use this matching to assign logical Pterms to physical nanowires in a nanoPLA (Chapter 38, Section 38.6) [17, 18]. Before considering defects, all the Pterm nanowires in the PLA are freely interchangeable. Each nanowire that implements a Pterm has a programmable diode between the input nanowires and the nanowire itself. If the diode is programmed into an off state, it disconnects the input from the nanowire Pterm. If the diode is in the on state, it connects the input to the nanowire, allowing it to participate in the AND that the Pterm is computing.

The most common defect anticipated in this technology is that the programmable diode is stuck in an off state—that is, it cannot be programmed into a valid on state. Consequently, a Pterm nanowire with a stuck-off diode at a

particular input location cannot be programmed to include that input in the AND it is performing.

A typical PLA will have 100 inputs, meaning each product-term nanowire is connected to 100 programmable diodes. A plausible failure rate for the product-term diodes is 5% ($P_d = 0.05$). If we demanded that each Pterm be defect free in order to use it, the yield of product terms would be:

$$P_{nwpterm}(100, 0.05) = (1 - 0.05)^{100} \approx 0.006 \tag{37.9}$$

However, since none of the product terms use all 100 inputs, the probability that a particular Pterm nanowire can support a logical Pterm is much higher. For example, if the Pterm only uses 10 inputs, then the probability that a particular Pterm nanowire can support it is:

$$P_{nwpterm}(10, 0.05) = (1 - 0.05)^{10} \approx 0.599 \tag{37.10}$$

Further, typical arrays will have 100 product-term nanowires. This suggests that, on average, this Pterm will be compatible with roughly 60 of the Pterm nanowires in the array—that is, the l_i for this Pterm will end up with compatibility edges to 60 r_j's in the bipartite matching graph described before.

As a result, DeHon and Naeimi [18] were able to demonstrate that we can tolerate stuck-off diode defects at $P_d = 0.05$ with no allocated spare nanowires. In other words, we can have $|L|$ as large as $|R|$ and, in practice, always find a complete matching for every PLA. This is true even though the probability of a perfect nanowire is below 1 percent (equation 37.9), suggesting that most arrays of 100 nanowires contain no perfect Pterm nanowires.

This strategy follows the defect map model and does demand component-specific mapping. Nonetheless, the required mapping is local (see the Local sparing section) and can be fast. Naeimi and DeHon [17] demonstrate the results quoted previously using a greedy, linear-time assignment algorithm rather than the slower, optimal algorithm. Further, if it is possible to test the compatibility of each Pterm as part of the trial assignment, it is not necessary to know the defect map prior to mapping.

FPGA component level

It is also possible to apply this matching idea at the component level. Here, the substitutable unit is an entire FPGA component. Unused resources will be switches, wires, and LUTs that are not used by a specific user design. Certainly, if the specific design does not fill the logic blocks in the component, there will be unused logic blocks whose failure may be irrelevant to the proper functioning of the design. Even if the specific design uses all the logic blocks, it will not use all the wires or all the features of every logic block. So, as long as the defects in the component do not intersect with the resources used by an particular FPGA configuration, the FPGA can perfectly support the configuration.

Xilinx's EasyPath series is one manifestation of this idea. At a reduced cost compared to perfect FPGAs, Xilinx sells FPGAs that are only guaranteed to

work with a particular user design, or a particular set of user designs. The user provides their designs, and Xilinx checks to see whether any of their defective devices will successfully implement those designs. Here, Xilinx's resource set, *R*, is the nonperfect FPGAs that do not have defects in the nonrepairable portion of the logic. The logical set, *L*, is the set of customer designs destined for EasyPath. Xilinx effectively performs the matching and then supplies each customer with FPGA components compatible with their respective designs.

Hyder and Wawrzynek [19] demonstrate that the same idea can be exploited in board-level FPGA systems. Here, their resource set, *R*, is the set of FPGAs on a particular board with multiple FPGAs. Their logical set is the set of FPGA configurations intended for the board. If all the FPGAs on the board were interchangeable, this would also reduce to the previous simple matching problem. However, in practice, the FPGAs on a board typically have different connections. This provides an additional set of topological constraints that must be considered along with resource compatibility during assignment. Rather than creating and maintaining a full defect map of each FPGA in the system, they also use application-specific testing (e.g., Tahoori [20]) to determine whether a particular FPGA configuration is compatible with a specific component on the FPGA board.

37.3 TRANSIENT FAULT TOLERANCE

Recall that transient faults are randomly occurring, temporary deviations from the correct circuit behavior. It is not possible to test for transient faults and configure around them as we did with defects. The impact of a transient fault depends on the structure of the logic and the location of the transient fault. The fault may be masked (hidden by downstream gates that are not currently sensitive to this input), may simply affect the circuit output temporarily, or may corrupt state so that the effect of the transient error persists in the computation long after the fault has occurred. Examples include the following:

- If both inputs to an OR gate should be 1, but one of the inputs is erroneously 0, the output of the OR gate will still have the correct value.
- If the transient fault impacts the combinational output from a circuit, only the output on that cycle is affected; subsequent output cycles will be correct until another transient fault occurs.
- If the transient fault results in the circuit incorrectly calculating the next state transition in a finite-state machine (FSM), the computation may proceed in the incorrect state for an indefinite period of time.

To deal with the general case where transient faults impact the observable behavior of the computation, we must be able to prevent the errors from propagating into critical state or to observable outputs from the computation. This demands that we add or exploit some form of redundancy in the calculation to detect or correct errors as they occur. This section reviews two general

approaches to transient fault tolerance: feedforward correction (Section 37.3.1) and rollback error recovery (Section 37.3.2).

37.3.1 Feedforward Correction

One common strategy to tolerate transient faults is to provide adequate redundancy to correct any errors that occur. This allows the computation to continue without interruption. The simplest example of this redundancy is replication. That is, we arrange to perform the intended computation R times and vote on the result, using the majority result as the value allowed to update state or to be sent to the output. The smallest example uses $R = 3$ and is known as triple modular redundancy (TMR) (see Figure 37.7). In general, for there to be a clear majority, R must be odd, and a system with R replicas can tolerate at least $\frac{R-1}{2}$ simultaneous transient faults. We can perform the multiple calculations either in space, by concurrently placing R copies of the computation on the reconfigurable array, or in time, by performing the computation multiple times on the same datapath.

In the simple design in Figure 37.7, a failure in the voter may still corrupt the computation. This can be treated similarly to nonrepairable area in defect-tolerance schemes:

- If the computation is large compared to the voter, the probability of voter failure may be sufficiently small so that it is acceptable.
- The voter can be implemented in a more reliable technology, such as a coarser-grained feature size.
- The voter can be replicated as well. For example, von Neumann [21] and Pippenger [22] showed that one can tolerate high transient fault rates (up to 0.4 percent) using a gate-level TMR scheme with replicated voters.

TMR strategies have been applied to Xilinx's Virtex series [23]. Rollins et al. [24] evaluate various TMR schemes on Virtex components, including strategies with replicated voters and replicated clock distribution.

A key design choice in modular redundancy schemes is the granularity at which voting occurs. At the coarsest grain, the entire computational circuit could be the unit of replication and voting. At the opposite extreme, we can replicate and vote individual gates as the Von Neumann design suggests. The appropriate choice will balance area overhead and fault rate. From an area

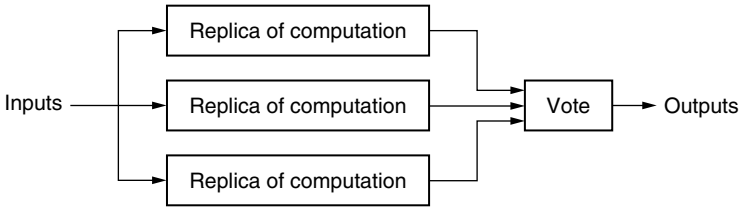

FIGURE 37.7 ■ A simple TMR design.

overhead standpoint, we would prefer to vote on large blocks; this allows the area of the voters to be amortized across large logic blocks so that the total area grows roughly as the replication factor, R. From an area overhead standpoint, we also want to keep R low. From a reliability standpoint, we want to make it sufficiently unlikely that more than $\frac{R-1}{2}$ replicas are corrupted by transient errors in a single cycle. Similar to defects (equation 37.4), the failure rate of a computation, and hence a replica, scales with the number of devices in the computation and the transient fault rate per device; consequently, we want to scale the unit of replication down as fault rate increases to achieve a target reliability with low R.

Memory

A common form of feedforward correction is in use today in memories. Memories have traditionally been the most fault-sensitive portions of components because: (1) A value in a memory may not be updated for a large number of cycles; as such, memories integrate faults over many cycles. (2) Memories are optimized for density; as such, they often have low capacitance and drive strength, making them more susceptible to errors.

We could simply replicate memories, storing each value in R memories or memory slots and voting the results. However, over the years information theory research has developed clever encoding schemes that are much more efficient for protecting groups of data bits than simple replication [25,26]. For example, DRAMs used in main memory applications generally tolerate a single-bit fault in a 64-bit data-word using a 72-bit error correcting code. Like the nonrepairable area in DRAMs, the error correcting circuitry in memories is generally built from coarser technology than the RAM memory array and is assumed to be fault free.

37.3.2 Rollback Error Recovery

An alternative technique to feedforward correction is to simply detect when errors occur and repeat the computation when an error is detected. We can detect errors with less redundancy than we need to correct errors (e.g., two copies of a computation are sufficient to detect a single error, while three are required for correction); consequently, detection schemes generally require lower overhead than feedforward correction schemes. If fault rates are low, it is uncommon for errors to occur in the logic. In most cycles, no errors occur and the normal computation proceeds uninterrupted. In the uncommon case in which a transient fault does occur, we stop processing and repeat the computation in time without additional hardware. With reasonably low transient-fault rates, it is highly unlikely that repeated computation will also be in error; in any case, detection guards against errors in the repeated computation as well.

To be viable, the rollback technique demands that the application tolerate stalls in computation during rollback. This is easily accommodated in streaming models (Chapter 5, Section 5.1.3) that exploit data-presence signaling (see Data

presence subsection of Section 5.2.1) to tolerate variable timing for operator implementations. When detection and rollback are performed on an operator level, stream buffers between operator datapaths can isolate and minimize the performance impact of rollback.

Detection

To detect errors we use some form of redundancy. Again, this can be either temporal or spatial redundancy.

To minimize the performance impact, we can employ a *concurrent-error detection* (CED) technique—that is, in parallel with the normal logic, we compute some additional function or property of the output (see Figure 37.8). We continuously check consistency between the logical output and this concurrent calculation. If the concurrent calculation ever disagrees with the base computation, this means there is an error in the logic.

In the simplest case, the parallel function could be a duplicate copy of the intended logic (see Figure 37.8(b)). Checking then consists of verifying that the two computations obtained the equivalent results. However, it is often possible to avoid recomputing the entire function and, instead, compute a property of the output, such as its parity (see Figure 37.8(c)) [27].

The choice of detection granularity is based on the same basic considerations discussed before for feedforward replica granularity. Larger blocks can amortize out comparison overhead but will increase block error rates and hence the rate of rollback. For a given fault rate, we reduce comparison block granularity until the rollback rate is sufficiently low so that it has little impact on system throughput.

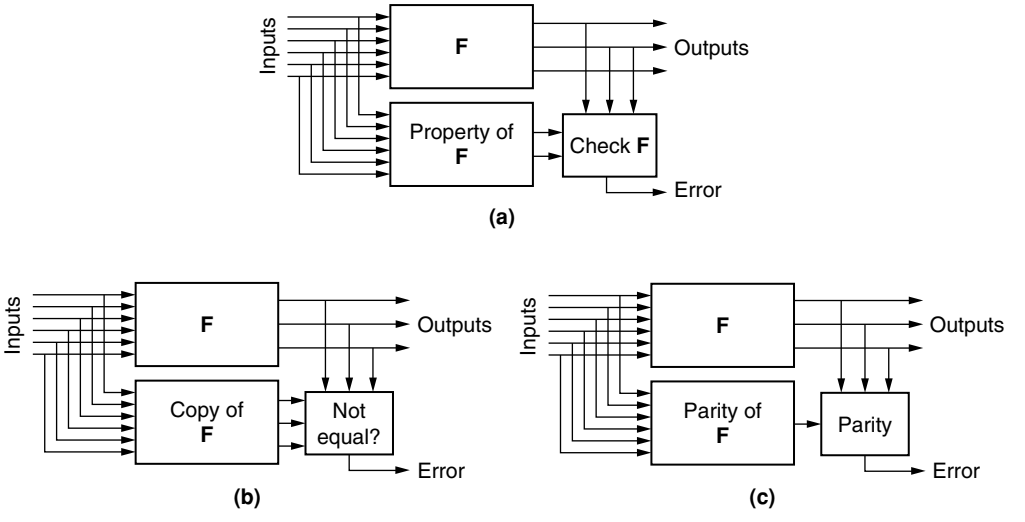

FIGURE 37.8 ■ A concurrent error-detection strategy and options: (a) generic formulation, (b) duplication, and (c) parity.

Recovery

When we do detect an error, it is necessary to repeat the computation. This typically means making sure to preserve the inputs to a computation until we can be certain that we have reliably produced a correct result. Conceptually, we read inputs and current state, calculate outputs, detect errors, then produce outputs and save state if no errors are detected. In practice, we often want to pipeline this computation so that we detect errors from a previous cycle while the computation continues, and we may not save state to a reliable storage on every calculation. However, even in sequential cases, it may be more efficient to perform a sequence of computations between error checks.

A common idiom is to periodically store, or *snapshot*, state to reliable memory, store inputs as they arrive into reliable memory, perform a series of data computations, and store results to reliable memory. If no errors are detected between snapshots, then we continue to compute with the new state and discard the inputs used to produce it. If errors are detected, we discard the new state, restore the old state, and rerun the computation using the inputs stored in reliable memory. As noted earlier in the Memory subsection, we have particularly compact techniques for storing data reliably in fault-prone memories; this efficient protection of memories allows rollback recovery techniques to be robust and efficient.

In streaming systems, we already have FIFO streams of data between operators. We can exploit these memories to support rollback and retry. Rather than discarding the data as soon as the operator reads it, we keep it in the FIFO but advance the head pointer past it. If the operator needs to rollback, we effectively reset the head pointer in the FIFO to recover the data for reexecution. When an output is correctly produced and stored in an output FIFO, we can then discard the associated inputs from the input FIFOs. For operators that have bounded depth from input to output, we typically know that we can discard an input set for every output produced.

Communications

Data transmission between two distant points, especially when it involves crossing between chips and computers, is highly susceptible to external noise (e.g., crosstalk from nearby wires, power supply noise, clock jitter, interference from RF devices). As such, for a long time we have protected communication channels with redundancy. As with memories, we simply need to reliably deliver the data sent to the destination.

Unlike memories, we do not necessarily need to guarantee that the correct data can be recovered from the potentially corrupted data that arrive at the destination. When the data are corrupted in transmission, it suffices to detect the error. The sender holds onto a copy of the data until the receiver indicates they have been successfully received. When an error is detected, the sender can retransmit the data. The detection and retransmission are effectively a rollback technique.

When the error rates on the communication link are low, such that error detection is the uncommon event, this allows data to be protected with low overhead error-detecting codes, or *checksums*, instead of more expensive

error correcting codes. The Transmission Control Protocol (TCP) used for communication across the Internet includes packet checksums and retransmission when data fail to arrive error free at the intended destination [28].

37.4 LIFETIME DEFECTS

Over the lifetime of a component, the physical device will change and degrade, potentially introducing new defects into the device. Individual atomic bonds may break or metal may migrate, increasing the resistance of the path or even breaking a connection completely. Device characteristics may shift because of hot-carrier injection (e.g., [29,30]), NBTI (e.g., [31]), or even accumulated radiation doses (e.g., [32, 33]). These effects become more acute as feature sizes shrink. To maintain correct operation, we must detect the errors (Section 37.4.1) and repair them (Section 37.4.2) during the lifetime of the component.

37.4.1 Detection

One way to detect lifetime failures is to periodically retest the device—that is, we stop normal operation, run a testing routine (see the Testing subsection in Section 37.2.4), then resume normal operation if there are no errors. It can be an application-specific test, determining whether the FPGA can still support the user's mapping [20], or an application-independent test of the FPGA substrate. Application-specific tests have the advantage of both being more compact and ignoring new defects that do not impact the current design. Substrate tests may require additional computation to determine whether the newly defective devices will impact the design. While two consecutive, successful tests generally mean that the computation between these two points was correct, the component may begin producing errors at any time inside the interval between tests and the error will not be detected until the next test is run.

Testing can also be interleaved more directly with operation. In partially reconfigurable components (see Section 4.2.3), it is possible to reconfigure portions of a component while the rest of the component continues operating. This allows the reservation of a fraction of the component for testing. If we then arrange to change the specific portions of the component assigned to testing and operation over time, we can incrementally test the entire component without completely pulling it out of service (e.g., [34,35]).

In some scenarios, the component may need to stall operation during the partial reconfiguration, but the component only needs to stall for the reconfiguration period and not the entire testing period. When the total partial reconfiguration time is significantly shorter than the testing time, this can reduce the fraction of cycles the application must be removed from normal operation. This still means that we may not detect the presence of a new defect until long after it occurred and started corrupting data.

If it is necessary to detect an error immediately, we must employ one of the fault tolerance techniques reviewed in Section 37.3. CED (see the Detection

subsection in Section 37.3.2) can identify an error as soon as it occurs and stall computation. TMR (Section 37.3.1) can continue correct operation if only a single replica is affected; the TMR scheme can be augmented to signal higher-level control mechanisms when the voters detect disagreement.

37.4.2 Repair

Once a new error has occurred, we can repeat global (see the Global sparing subsection in Section 37.2.4) or local mapping (see the Local sparing subsection in Section 37.2.4) to avoid the new error. However, since the new defect map is most likely to differ from the old defect map by only one or a few defects, it is often easier and faster to incrementally repair the configuration. In local mapping schemes, we only need to perform local remapping in the interchangeable region(s) where the new defect(s) have occurred. This may mean that we only need to move LUTs in a single cluster, wires in channel, or remap a single tile. Even in global schemes the incremental work required may be modest. Lakamraju and Tessier [36] show that incrementally rerouting connections severed by new lifetime defects can be orders of magnitude faster than performing a complete reroute from scratch.

A rollback scheme (Section 37.3.2) can stall execution during the repair. A replicated, feedforward scheme (Section 37.3.1) with partial reconfiguration may be able to continue operating on the functional replicas while the newly defective replica is being repaired.

Lifetime repair strategies depend on the ability to perform defect mapping and reconfiguration. Consequently, the perfect component model cannot support lifetime repair. Even if the component retains spare redundancy, redundancy and remapping mechanisms are not exposed to the user for in-field use.

37.5 CONFIGURATION UPSETS

Many reconfigurable components, such as FPGAs, rely on volatile memory cells to hold their configuration, typically static memory cells (e.g., SRAM). Dynamic memory cells have long had to cope with upsets from ionizing particles (e.g., α-particles). As the feature sizes shrink, even static RAM cells can be upset by ionizing particles (e.g., Harel et al. [37]). In storage applications, we can typically cope with memory soft errors using error correcting codes (see the Memory subsection in Section 37.3.1) so that bit upsets can be detected and corrected. However, in reconfigurable components, we use the memory cells directly and continuously as configuration bits to define logic and interconnect. Upsets of these configuration memories will change, and potentially corrupt, the logic operation.

Unfortunately, although memories can amortize the cost of a large error correction unit across a deep memory, FPGA configurations are shallow (i.e., $N_{instr} = 1$); an error correction scheme similar to DRAM memories would end up being as large as or larger than the configuration memory it protects. Data

and projections from Quinn and Graham [38] suggest that ionizing radiation upsets can be a real concern for current, large FPGA-based systems and will be an ongoing concern even for modest systems as capacity continues to increase.

Because these are transient upsets of configuration memories, they can be corrected simply by reloading the correct bitstream once we detect that the bitstream has been corrupted. Logic corruption can be detected using any of the strategies described earlier for lifetime defects (Section 37.4.1). Alternatively, we can check the bitstream directly for errors. That is, we can compute a checksum for the correct bitstream, read the bitstream back periodically, compute the checksum of the readback bitstream, and compare it to the intended bitstream checksum to detect when errors have occurred. When an error has occurred, the bitstream can be reloaded [38, 39]. Like interleaved testing, bitstream readback introduces a latency, which can be seconds long, between configuration corruption and correction. If the application can tolerate infrequent corruption, this may be acceptable.

Asadi and Tahoori [40] detail a rollback scheme for tolerating configuration upsets. Pratt et al. [41] use TMR and partial TMR schemes to tolerate configuration upsets; their partial TMR scheme uses less area than a full TMR scheme in cases where it is acceptable for the outputs to be erroneous for a number of cycles as long as the state is protected so that the results return to the correct values when the configuration is repaired.

37.6 OUTLOOK

The regularity in reconfigurable arrays, coupled with the resource configurability they already possess, allow these architectures to tolerate defects. As features shrink and defect rates increase, all devices, including ASICs, are likely to need some level of regularity and configurability; this will be one factor that serves to narrow the density and cost gap between FPGAs and ASICs. Further, at increased defect rates, it will likely make sense to ship components with defects and defect maps. Since each component will be different, some form of component-specific mapping will be necessary.

Transient upsets and lifetime defects further suggest that we should continuously monitor the computation to detect errors. To tolerate lifetime defects, repair will become part of the support system for components throughout their operational lifetime. Increasing defect rates further drive us toward architectures with finer-grained substitutable units. FPGAs are already fairly fine grained, with each bit-processing operator potentially serving as a substitutable unit, but finer-grained architectures that substitute individual wires, Pterms, or LUTs may be necessary to exploit the most aggressive technologies.

References

[1] S. E. Schuster. Multiple word/bit line redundancy for semiconductor memories. *IEEE Journal of Solid State Circuits* 13(5), 1978.

[2] B. Keeth, R. J. Baker. *DRAM Circuit Design: A Tutorial*. Microelectronic Systems, IEEE Press, 2001.
[3] J. Bernoulli. *Ars Conjectandi*. Impensis thurnisiorum, fratrum, Basel, Switzerland, 1713.
[4] A. W. Drake. *Fundamentals of Applied Probability Theory*, McGraw-Hill, 1988.
[5] A. DeHon. Law of large numbers system design. *Nano, Quantum and Molecular Computing: Implications to High Level Design and Validation*, S. K. Shukla, R. I. Bahar (eds.), Kluwer Academic, 2004.
[6] W. K. Huang, F. J. Meyer, X.-T. Chen, F. Lombardi. Testing configurable LUT-based FPGAs. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 6(2), 1998.
[7] W. B. Culbertson, R. Amerson, R. Carter, P. Kuekes, G. Snider. Defect tolerance on the TERAMAC custom computer. *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, 1997.
[8] M. Mishra, S. C. Goldstein. Defect tolerance at the end of the roadmap. *Proceedings of the International Test Conference (ITC)*, 2003.
[9] M. Mishra, S. C. Goldstein. Defect tolerance at the end of the roadmap. *Nano, Quantum and Molecular Computing: Implications to High Level Design and Validation*, S. K. Shukla, R. I. Bahar (Eds.), Kluwer Academic, 2004.
[10] R. G. Cliff, R. Raman, S. T. Reddy. Programmable logic devices with spare circuits for replacement of defects. U.S. Patent number 5,434,514, July 18, 1995.
[11] C. McClintock, A. L. Lee, R. G. Cliff. Redundancy circuitry for logic circuits. U.S. Patent number 6,034,536, March 7, 2000.
[12] J. Lach, W. H. Mangione-Smith, M. Potkonjak. Low overhead fault-tolerant FPGA systems. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 26(2), 1998.
[13] A. J. Yu, G. G. Lemieux. Defect-tolerant FPGA switch block and connection block with fine-grain redundancy for yield enhancement. *Proceedings of the International Conference on Field-Programmable Logic and Applications*, 2005.
[14] A. J. Yu, G. G. Lemieux. FPGA defect tolerance: Impact of granularity. *Proceedings of the International Conference on Field-Programmable Technology*, 2005.
[15] T. Cormen, C. Leiserson, R. Rivest. *Introduction to Algorithms*. MIT Press, 1990.
[16] J. E. Hopcroft, R. M. Karp. An $n^{2.5}$ algorithm for maximum matching in bipartite graphs. *SIAM Journal on Computing* 2(4), 1973.
[17] H. Naeimi, A. DeHon. A greedy algorithm for tolerating defective crosspoints in nanoPLA design. *Proceedings of the International Conference on Field-Programmable Technology*, IEEE, 2004.
[18] A. DeHon, H. Naeimi. Seven strategies for tolerating highly defective fabrication. *IEEE Design and Test of Computers* 22(4), 2005.
[19] Z. Hyder, J. Wawrzynek. Defect tolerance in multiple-FPGA systems. *Proceedings of the International Conference on Field-Programmable Logic and Applications*, 2005.
[20] M. B. Tahoori. Application-dependent testing of FPGAs. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 14(9), 2006.
[21] J. von Neumann. Probabilistic logic and the synthesis of reliable organisms from unreliable components. *Automata Studies* C. Shannon, J. McCarthy (ed.), Princeton University Press, 1956.
[22] N. Pippenger. Developments in "the synthesis of reliable organisms from unreliable components." *Proceedings of the Symposia of Pure Mathematics* 50, 1990.
[23] C. Carmichael. *Triple Module Redundancy Design Techniques for Virtex FPGAs*. San Jose, 2006 (XAPP 197—*http://www.xilinx.com/bvdocs/appnotes/xapp197.pdf*).

[24] N. Rollins, M. Wirthlin, P. Graham, M. Caffrey. Evaluating TMR techniques in the presence of single event upsets. *Proceedings of the International Conference on Military and Aerospace Programmable*, 2003.
[25] G. C. Clark Jr., J. B. Cain. *Error-Correction Coding for Digital Communications*, Plenum Press, 1981.
[26] R. J. McEliece. *The Theory of Information and Coding*, Cambridge University Press, 2002.
[27] S. Mitra, E. J. McCluskey. Which concurrent error detection scheme to choose? *Proceedings of the International Test Conference*, 2000.
[28] J. Postel (ed.). Transmission Control Protocol—DARPA Internet Program Protocol Specification, RFC 793, Information Sciences Institute, University of Southern California, Marina del Rey, 1981.
[29] E. Takeda, N. Suzuki, T. Hagiwara. Device performance degradation to hot-carrier injection at energies below the Si-SiO2 energy barrier. *Proceedings of the International Electron Devices Meeting*, 1983.
[30] S.-H. Renn, C. Raynaud, J.-L. Pelloie, F. Balestra. A thorough investigation of the degradation induced by hot-carrier injection in deep submicron N- and P-channel partially and fully depleted unibond and SIMOX MOSFETs. *IEEE Transactions on Electron Devices* 45(10), 1998.
[31] D. K. Schroder, J. A. Babcock. Negative bias temperature instability: Road to cross in deep submicron silicon semiconductor manufacturing, *Journal of Applied Physics* 94(1), 2003.
[32] J. Osborn, R. Lacoe, D. Mayer, G. Yabiku. Total dose hardness of three commercial CMOS microelectronics foundries. *Proceedings of the European Conference on Radiation and Its Effects on Components and Systems*, 1997.
[33] C. Brothers, R. Pugh, P. Duggan, J. Chavez, D. Schepis, D. Yee, S. Wu. Total-dose and SEU characterization of 0.25 micron CMOS/SOI integrated circuit memory technologies. *IEEE Transactions on Nuclear Science* 44(6) 1997.
[34] J. Emmert, C. Stroud, B. Skaggs, M. Abramovici. Dynamic fault tolerance in FPGAs via partial reconfiguration. *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, 2000.
[35] S. K. Sinha, P. M. Kamarchik, S. C. Goldstein. Tunable fault tolerance for runtime reconfigurable architectures. *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, 2000.
[36] V. Lakamraju, R. Tessier. Tolerating operational faults in cluster-based FPGAs. *Proceedings of the International Symposium on Field-Programmable Gate Arrays*, 2000.
[37] S. Harel, J. Maiz, M. Alavi, K. Mistry, S. Walsta, C. Dai Impact of CMOS process scaling and SOI on the soft error rates of logic processes. *Proceedings of Symposium on VLSI Digest of Technology Papers*, 2001.
[38] H. Quinn, P. Graham. Terrestrial-based radiation upsets: A cautionary tale. *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, 2005.
[39] C. Carmichael, M. Caffrey, A. Salazar. *Correcting Single-Event Upsets Through Virtex Partial Configuration*. Xilinx, Inc., San Jose, 2000 (XAPP 216—*http://www.xilinx.com/bvdocs/appnotes/xapp216.pdf*).
[40] G.-H. Asadi, M. B. Tahoori. Soft error mitigation for SRAM-based FPGAs. *Proceedings of the VLSI Test Symposium*, 2005.
[41] B. Pratt, M. Caffrey, P. Graham, K. Morgan, M. Wirthlin. Improving FPGA design robustness with partial TMR. *Proceedings of the IEEE International Reliability Physics Symposium*, 2006.