Appearing in *IEEE International Conference on Field-Programmable Technology (FPT 2013), December 9–11, 2013*

Exploiting Partially Defective LUTs: Why You Don't Need Perfect Fabrication

André DeHon
Department of Electrical and Systems Engineering
University of Pennsylvania, Philadelphia, PA 19104
Email: andre@ieee.org

Nikil Mehta
Tabula, Inc.
3250 Olcott St Ste 300, Santa Clara, CA 95054
Email: nmehta@tabula.com

***Abstract*—Shrinking integrated circuit feature sizes lead to increased variation and higher defect rates. Prior work has shown how to tolerate the failure of entire LUTs and how to tolerate failures and high variation in interconnect. We show how to use LUTs even when they are partially defective—a form of fine-grained defect tolerance. We characterize the defect tolerance of a range of mapping strategies for defective LUTs, including LUT swapping in a cluster, input permutation, input polarity selection, defect-aware packing, and defect-aware placement. By tolerating partially defective LUTs, we show that, even without allocating dedicated spare LUTs, it is possible to achieve near perfect yield with cluster local remapping when roughly 1% of the LUT multiplexers fail to switch. With full, defect-aware placement, this can increase to 10–25% with just a few extra rows and columns. In contrast, substitution of perfect LUTs to dedicated spares only tolerates failure rates of 0.01–0.05%.**

I. Introduction

As integrated circuit feature sizes continue to shrink, we will see increasing levels of transistor variability and increasing rates of device failure. To achieve net benefits from future technologies, we must find inexpensive ways to exploit the resulting, imperfect components. Prior work (Sec. II-B) has shown that FPGAs provide the opportunity to avoid defective logic and interconnect elements by sparing at the level of interconnect segments [1] and LUTs [2]. This, creates a discrepancy in granularity. An interconnect segment only needs a few transistors and memory bits to be functional, whereas a k-LUT requires 2^k memory bits and $4 \times \left(2^k - 1\right)$ multiplexer transistors (Fig. 1). Assuming 6 transistor memory cells, this is over 150 transistors for a 4-LUT. If we demand that we only use defect-free LUTs, the rate of LUT failure will be much higher than the rate of interconnect failure, meaning LUT yield could become the limiter. This could drive us to smaller LUT sizes (40 transistors for a 2-LUT), but we know that small LUTs are not the most efficient for implementing logic [3].

By using partially defective LUTs, we can effectively achieve finer-grained resource sparing, allowing us to continue to use LUT sizes that are efficient for logic. In particular, we show that the most common, variation-induced failures in LUTs is for the LUT multiplexers to fail to switch but still be able to hold a constant value (*constant multiplexer* failure, Sec. III) and that most logic functions can tolerate these failures (Sec. IV). Furthermore, we show transforms that allow us to change how the logic function is mapped to the LUT multiplexer, increasing the probability that the LUT function can be successfully implemented on the partially defective LUT. We introduce, catalog, and characterize a range of remapping strategies (Sec. VI) that tolerate fine-grained LUT multiplexer failures. The simplest schemes are limited to local exchanges within the cluster that can be performed quickly, while the most powerful schemes exploit full LUT placement.

Our novel contributions include:

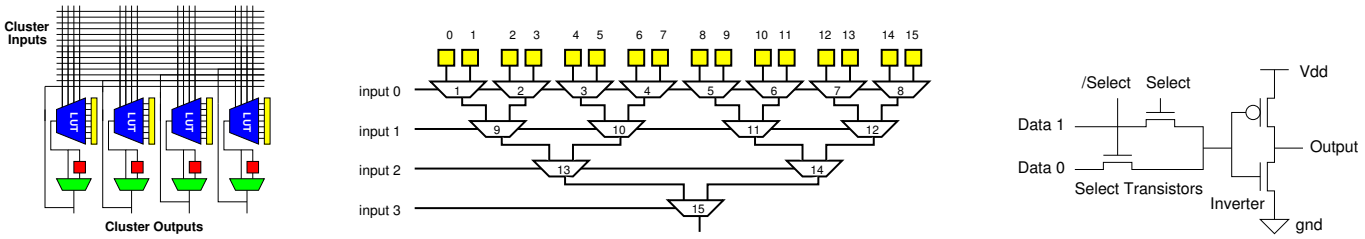
1) Characterizing variation-induced LUT failure modes, identifying the *constant multiplexer* failure model, and formulating the *partially defective LUT mapping* problem for the constant-multiplexer failure model (Sec. III)
2) Identifying mitigating transformations to tolerate constant multiplexer failures in an island-style cluster FPGA architecture (Sec. IV)
3) Cataloging a range of mapping strategies that exploit these transformations (Sec. VI)
4) Introducing the first defect-aware clustering algorithm (Sec. VI-C)
5) Quantifying the constant multiplexer failure defect tolerance achievable with a range of mitigation techniques (Sec. VII)

II. Background

A. FPGA Logic

The logic in FPGAs is implemented with a small LookUp-Table (LUT). For a k-input logic function, this is simply a k-input, 1-output memory and is typically implemented as 2^k LUT configuration memory bits followed by a 2^k:1 multiplexer (Fig. 1b). Any logic function of k inputs can be implemented by storing its truth table into the LUT configuration memory bits. The impacts of LUT size on delay, energy, and area has been an area of active research, determining that 4-LUTs are the most area [3] and energy [4] efficient, with larger LUTs (*e.g.*, 6-LUTs) providing the lowest latency [5]. Modern, commercial FPGAs (*e.g.*, [6]) make the LUTs decomposible and augment them with hardwired logic.

Rather than simply using a uniform mesh to interconnect individual LUTs, modern FPGAs will cluster a number of LUTs together into a logic block (Fig. 1a). The LUTs within a cluster share a limited number of inputs from the mesh interconnect and are internally connected with a local, intra-cluster crossbar. Connections within the cluster go through a single stage of switching and are faster than inter-cluster connections. Prior work has studied the impact of cluster size on area, delay, and energy of the FPGA (*e.g.*, [7], [4], [5]).



(a) Island-Style Clustered Logic Block

(b) 4-LUT Built from 2-input Multiplexers

(c) Transistor-Level Implementation for 2-Input Multiplexer Used in LUT

Fig. 1. FPGA Logic Block Composition

B. Prior Work on FPGA Defect and Variation Tolerance

A large set of prior work observes that FPGAs can tolerate defects by mapping to avoid the defective elements in a particular chip. Perhaps the earliest full-scale exploitation of this idea was Hewlett-Packard's TERAMAC [8], which used a combination of architecture design and full defect-aware placement and routing. The custom, TERAMAC FPGA architecture [9] was more richly interconnected than traditional FPGAs, leaving questions as to how much benefit came from the custom architecture and how much came from the defect-aware mapping. On the PLASMA chip, 10.4% of the 6-LUTs failed, a percentage over 3 times that of crossbar lines (3% defect rate) and 7 times that of crossbar buffers (1.5% defect rate), underscoring the concern that LUTs may fail at higher rates than interconnect elements.

1) Logic: Lach [10] first showed that FPGAs could pre-allocate spares and be locally reconfigured to avoid defective logic. He suggested reserving spares in an $m \times m$ region of an FPGA and providing a set of mappings to avoid any single or combination of defects within that region. Nonetheless, Lach only showed how to tolerate failures in logic. With the advent of the modern island-style cluster architecture, Lakamraja [2] showed how to tolerate logic faults by allocating a spare LUT in a cluster. Neither Lach nor Lakamraja evaluated the impact of using partially defective LUTs.

2) Interconnect: Lakamraja [2] also showed that interconnect failures can be tolerated by rerouting the design to avoid defective interconnect segments. In our previous work [1], we showed that it is necessary to tolerate defective switches to operate near the minimum-energy operating point for the high-variation technologies between now and the end of the semiconductor roadmap. Using delay-aware routing, we tolerate around 1% unusable interconnect switches, allowing FPGAs to operate at half the energy required for a delay-oblivious mapping. In that previous work, we focussed on interconnect switches, leaving LUT variation and defects for future work, which this paper now addresses.

III. LUT Failures

A. Multiplexer Model

For concreteness, we assume the LUTs are built as a tree of multiplexers (Fig. 1b). We further assume each 2-input multiplexer is implemented using 4 transistors as an NMOS only pass-gate multiplexer followed by an inverter (Fig. 1c). There are, of course, a variety of implementation options for the LUT multiplexer, including using complementary pass gates, flattening the multiplexers, and cascading multiple pass-gate stages before CMOS restoration. We choose the implementation in Fig. 1c since it is compact and simple to analyze, and we believe the high-level insight that arises from it will be valid across implementation options. Note that the individual 2-input multiplexer is comparable in complexity to the S-box switch in a direct-drive architecture [11] that was assumed in [1].

B. Multiplexer Failure

The most common failure mode in this model is that a LUT multiplexer will fail to switch in the case where one of its inputs is zero and the other is one, but it will correctly drive a zero or one on its output if its two data inputs are both zero or one. Consequently, we identify this particular failure mode as the *constant mulitplexer* failure mode. Intuitively, it is easy to understand why this failure mode is common by considering what happens when each of the transistors in multiplexer fails or switches slowly. If *any* of the 4 transistors in the multiplexer switches slowly, the output will always hold the correct value when the two data inputs are the same. The output of the multiplexer is charged after configuration, and it never needs to switch during operation. If one but, not both, of the NMOS select transistors switches fails to switch, the output of the pass-transistor stage cannot be controlled by the data input to that select transistor. However, since the other pass transistor can drive the output, it will charge the output to the correct output whenever it is "on", which happens to be the same value that would be driven by defective select transistor. So, the output will remain charged at the correct, constant value. If one or both of the NMOS select transistors are stuck "on", it will also correctly transfer the non-changing, identical data inputs to the multiplexer output. If an inverter transistor fails completely such that it can only drive high or only drive low, the multiplexer may only be able to produce a particular constant. For variation-induced failures, these cases are significantly less common as described below.

We abstract the above observation into a *constant multiplexer failure model*. Each multiplexer behaves in one of two ways: (1) *fully functional* – the multiplexer can proper pass either of its data inputs in a timely fashion; (2) *constant multiplexer* – the multiplexer performs properly only if both of its inputs are configured to always be the same value.

C. Variation-Induced Failure

We discovered the constant multiplexer failure model while studying the impact of high random V_{th} variation on FPGA

circuit elements [12]. Numerous manufacturing effects are statistical in nature and lead to high variation in transistor characteristics. In particular, random dopant fluctuation [13], line-edge roughness [14], and fluctuations in critical dimension such as oxide thickness [15], lead to high variation in the V_{th} of the transistor [16]. As such, in modern processes, V_{th} is best modeled as a Gaussian random variable with a mean and variation, $\sigma_{V_{th}}$ [17]. In Table DESN10, the International Technology Roadmap for Semiconductors (ITRS) [18] reports the expected variation that will be seen by a minimum sized device in future processes. The value reported in the ITRS table is $3\sigma_{V_{th}}$ and suggests that we are already seeing one sigma variation around 14% of the nominal V_{th} value and that this may grow to 26% over the next decade. This variation can be reduced by increasing the size (width, W or length, L) of the transistors in the circuit (Eq. 1), which vendors already do.

$$\sigma_{V_{th}} \propto \frac{1}{\sqrt{WL}} \tag{1}$$

This *reverse-scaling* of feature sizes increases area and energy in devices. If we can tolerate high variation, we can reduce the necessary reverse-scaling at more advanced technologies.

High variation means we will see a large range of V_{th} values in the individual transistors in a design. The Gaussian distributions suggest that one in 1000 transistors will have a V_{th} outside of $V_{th} \pm 3\sigma_{V_{th}}$ and one in a million will have V_{th} outside of $V_{th} \pm 5\sigma_{V_{th}}$. With over 150 transistors in a 4-LUT, we should expect to see LUT transistors that are $3\sigma_{V_{th}}$ out in most clusters and transistors that are 5–6$\sigma_{V_{th}}$ in 10,000–1,000,000 LUT FPGAs.

Since transistor drive current in the saturation region is driven by the *difference* between the gate voltage and V_{th} (Eq. 2), variation in V_{th} can result in low drive current and hence slow transistors (Eq. 3).

$$I_{dsat} = W v_{sat} C_{ox} \left(V_{gs} - V_{th} - \frac{V_{d,sat}}{2} \right)^{\gamma} \tag{2}$$

$$\tau_p = C V_{dd} / I_{dsat} \tag{3}$$

When V_{gs} drops below V_{th} the transistor moves into the subthreshold region where current is small but exponentially dependent on $V_{gs} - V_{th}$. For normal FPGA operation, we use this as the off-region for the devices. V_{gs} will be at most the supply voltage, V_{dd} for the logic. When using NMOS pass transistors, the output of the pass transistor can drive at most $V_{gs} = V_{dd} - V_{th}$ into the inverter. This potentially has a composite effect on the drive speed of the inverter, lowering its V_{gs} when its V_{th} may also be raised. For low V_{dd} and extreme variation, it is even possible to have inverters where the "on" drive current, I_{dsat}, for the PMOS or NMOS transistor is lower than the "off" current of the complementary transistor. In such cases, the inverter will never switch.

Raising the operating voltage, V_{dd}, can reduce the sensitivity to variation (Fig. 2) at the cost of higher switching energy since switching energy scales as CV^2. Since today's designs are energy limited, it is desirable to lower V_{dd} rather than to raise it. The most energy efficient operating points are at or below the threshold voltage [19]. However, if we cannot tolerate the failures that result from high variation, it will not be viable to operate at these energy-desirable operating points.

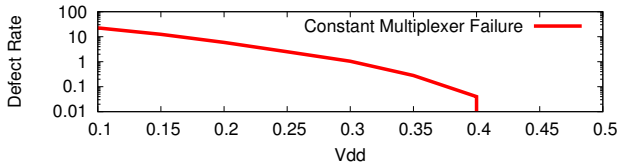


Fig. 2. LUT Multiplexer Failure Rates vs. V_{dd} Using 22 nm Low Power PTM Model Assuming $\sigma_{V_{th}} = 12\%$

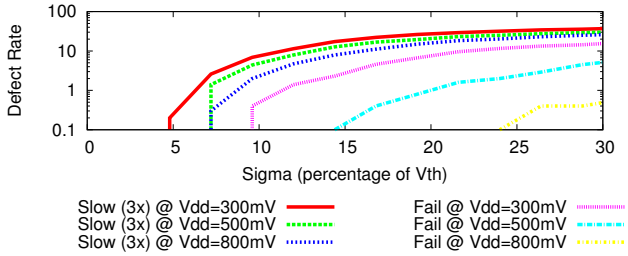


Fig. 3. LUT Multiplexer Failure Rates vs. Variation σ Using 22 nm Low Power PTM Model

To understand the impact of variation on the LUT multiplexer shown in Fig. 1c, we performed a series of Monte Carlo SPICE experiments on the multiplexer [12]. We used the predictive technology models (PTM) from Arizona State University [20] to model the transistors. The Monte Carlo simulation selects the V_{th} for each of the four transistors in the multiplexer independently from the Gaussian distribution according to a particular $\sigma_{V_{th}}$. At each $\sigma_{V_{th}}$, we generated 10,000 4-transistor V_{th} tuples and simulated the circuit in SPICE to determine its operational characteristics, failure modes, and speed of operation. Over the 10,000 samples, we saw no failures when the data inputs to the multiplexer were both zeros or ones. Fig. 2 shows how constant multiplexer failures increase with decreasing voltage. Fig. 3 shows how this failure increases with increasing $\sigma_{V_{th}}$ to capture both the effects of technology scaling that results in increased $\sigma_{V_{th}}$ at smaller feature sizes and transistor sizing (Eq. 1) that also results in increased variation at smaller sizes. "Slow (3x)" in Fig. 3 indicates the multiplexer switched at three times or more the nominal delay.

D. Configuration Robustness

SRAM failures due to variation are already a yield concern in industry. In normal memory arrays, the most likely SRAM failures modes under variation are read and access time failures [21]. However, FPGA SRAM configuration bits are not loaded by a large, common bitline and consequently are never read in the way SRAM bits in an array are. Consequently, they cannot have read related failures. Write failures can be avoided by overdriving the write voltage to the cell, which is reasonable since this only occurs during configuration and can be performed slowly compared to operation. Hold failures are the least likely of the common SRAM failure modes, and they may occur in FPGA SRAMs if the threshold voltages on the memory cell is such that the cell cannot hold a proper value—that is, one or both of the static restoration inverters fails to switch. We performed 10,000 Monte Carlo simulations for 6-T SRAM cells in the 22 nm Low Power PTM model and saw no failures for V_{dd} of 150mV or higher. At 100mV, we saw 2

failures, suggesting the failure rate is three orders of magnitude lower than the constant multiplexer failure rate [12].

E. Partially Defective LUT Mapping

We characterize the defect syndrome, D, of every LUT by a bit vector with one bit per multiplexer in the LUT (Fig. 1b). A one bit represents a constant multiplexer failure, while a zero bit represents a fully functional multiplexer. A LUT function F can tolerate a defective multiplexer j when all the configuration bits $F[i]$ that may pass through the multiplexer are either zero or one. For the numbering in Fig. 1b:

$$Tol(F,j) \equiv \begin{cases} 1 \le j < 9: & F[2j-1] = F[2j-2] \\ 9 \le j < 13: & F[4j-37] = F[4j-38] \\ & = F[4j-39] = F[4j-40] \\ j \in \{13,14\}: & F[8(j-13)-1] \\ & \vdots \\ & = F[8(j-13)-8] \\ j = 15: & F[0] = F[1] = \ldots = F[15] \end{cases}$$

A LUT function F tolerates a defect syndrome D when it tolerates all the multiplexer failures, $D[j] = 1$.

$$\begin{aligned} Tolerate(F,D) \quad = \quad & ((D[1]=0) \text{ OR } Tol(F,1)) \text{ AND} \\ & ((D[2]=0) \text{ OR } Tol(F,2)) \text{ AND} \\ & \vdots \\ & ((D[15]=0) \text{ OR } Tol(F,15)) \end{aligned} \quad (4)$$

The *partially defective LUT mapping problem* is to identify a mapping, π, that uniquely maps each LUT m in the design to a different physical LUT q on the chip with defect syndrome, D_q, such that $Tolerate(F_m, D_{\pi(m)})$.

IV. Opportunity

Our key point of leverage is that most functions do not need all of the multiplexers to be fully functional. Many common functions do have paired zeros and ones in the LUT function configuration meaning that these functions can tolerate some constant multiplexers. Furthermore, we can often transform the mapping to control how tolerable constant multiplexers in the LUT function align with the constant multiplexers in the fabricated, partially defective LUT.

A 4-input AND gate is an example function that only needs four fully functional multiplexers. The LUT function for an AND will have 15 zeros in the LUT configuration bits and a single one. This means that only one multiplexer in each row of multiplexers (Fig. 1b) must be fully functional. All the other multiplexers will see constant inputs. Specifically, for an AND4, configuration bit 15 will hold a one, while all the other configuration bits will be zero. Multiplexers 8, 12, 14, and 15 must be functional, while the remaining multiplexers can be constant multiplexers. Of course, some functions are harder to map than others. The 4-input XOR, for example, will require that all fifteeen multiplexers be fully functional.

We can also change the way the LUT function is mapped to change where the constant multiplexers occur. For example, if we can invert the polarity of input 3 for the AND4, we change the location of the necessary fully functional multiplexers to 4, 10, 13, 15, allowing us to tolerate constant multiplexers in 8, 12, and 14. For functions with mixed polarity inputs, like $a \cdot \bar{b} \cdot c \cdot d$, input assignment will change the location of the non-zero in the LUT configuration and change the set of multiplexers that must be fully functional. For example, if we assign a to input 0, b to 1, c to 2, and d to 3, the one is in configuration bit 13, and the required fully functional multiplexers are 7, 12, 14, 15. If instead we assign a to 1, b to 3, c to 0, and d to 2, the one is in configuration bit 7 and the required fully functional multiplexers are 4, 10, 13, 15. To illustrate how these transformations allow LUT functions to tolerate more defects, Table I considers all 2^{2^4}=65536 LUT functions and counts the number that can tolerate a constant multiplexer in the specified position(s) without any transformations, with input permutation alone, with inversion alone, and with both input permutation and inversion.

TABLE I. LUT Functions Tolerant to Specified Defects

Constant Multiplexer(s)	Transformation: None	Input Permute	Input Invert	Permute & Invert
mux1	4096	4096	16926	16926
mux9	4096	15096	16926	16926
mux13	4096	11712	16926	16926
mux3+mux11	512	17408	1694	19968

Number in table is the number of LUT functions (out of a total of 2^{2^4}=65536) that are tolerant to *constant* multiplexers in the identified positions when using the specified transformation.

TABLE II. Tolerable Constant Multiplexers Distribution

Tolerable Constant Multiplexers	0	1	2	3	4	5	6	7	8	9	10	11
LUT Count (clma)	0	0	0	0	0	0	193	3	62	115	16	3619
LUT Count (des)	26	0	0	1	0	0	369	5	25	64	105	637

The examples of the 4-input AND and 4-input XOR gates illustrate that different LUT functions are more or less tolerant to constant multiplexers. We can roughly characterize the difficulty we will have in mapping a LUT function to a partially defective LUT by counting the number of constant tolerable multiplexers for each function after accounting for potential input inversions and permutations. To understand how well this rough characterization performs, we map the `des` design from the Toronto 20 benchmarch set [22] to 4-LUTs using ABC [23] and, for each mapped LUT function, we both count the number of tolerable constant multiplexers and determine the number of defect syndromes the LUT function can tolerate. Fig. 4 shows the high correlation between these counts. All non-constant LUT functions will require at least 4 non-constant multiplexers, so in the best case there are 11 tolerable constant multiplexers. Some require all 15 to function, so can tolerate no constant multiplexers. It is much less expensive to count tolerable constant multiplexers than to compare the LUT function with all defect syndromes, making this count a useful approximation for LUT hardness for use in mapping tools. Using the same ABC mapping of `des` and an ABC maping of `clma`, Table II shows the distribution of tolerable constant multiplexers for the LUT functions. Very few functions are completely intolerant to constant multiplexers, and most functions tolerate many constant multiplexers. The mean number of tolerable constant multiplexers is 10 for `clma` and 9 for `des`.

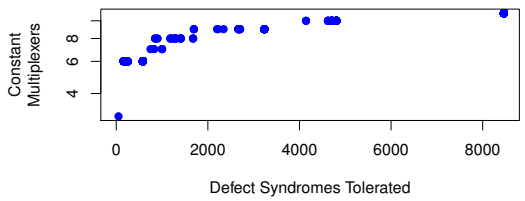


Fig. 4. Correlation of Cost Function that Counts Tolerable Constant Multiplexers and Count of Defect Syndromes Tolerated for `des`

V. Models

A. LUT Multiplexer Defects

For the experiments that follow, we generate constant multiplexer failures independently and randomly at a specified constant multiplexer failure rate of P_{const}, setting each LUT multiplexer 1–14 (Fig. 1b) in each LUT in the FPGA to either fully functional or constant. We do not generate failures for the output multiplexer, 15. Rather, we assume that this multiplexer is sized up (larger W, Eq. 1) so that it does not fail. This is the multiplexer that must drive into the interconnect, so it is typically sized up for performance. We sweep across different values of P_{const} to represent different technologies or sizing choices. In this manner, we generated defect maps for 100 "chips" at each P_{const} defect rate and used them across all benchmarks and mapping variants.

B. Architecture Model

For these experiments, we map to a baseline architecture with four 4-LUTs in each cluster and 4 input or output pads in each periphery I/O cluster. For the sparing experiments, we add a fifth 4-LUT to the logic clusters that we leave empty during clustering and potentially use during final mapping. This spare 4-LUT is identifical to the baseline 4-LUTs and its multiplexers fail at the same rates. These logic clusters have 10 inputs following the VPR5 release 4x4-LUT architecture (`k4-n4.xml`) [24] and the Toronto recommendations [7]. We do not model any specific population of the intra-LUT crossbar (Fig. 1a) or Connection-Box connections. For matching and permutation, we assume we have full freedom to move LUTs within the cluster and permute LUT inputs. That full freedom might not exist when using a depopulated intra-LUT crossbar [25], [26] or might demand that routing change to accommodate the transforms necessary for LUT mapping. As such, the full input transform cases serve as an upper bound on what might be possible, with a depopulated case coming somewhere between the transforms that assume full freedom and the more restricted transforms.

VI. Repair Algorithms

In this section, we catalog a range of mitigation optimizations to address the constant multiplexer mapping problem. We illustrate trends with the ABC mapping of the Toronto20 benchmark `clma` on the four 4-LUT per cluster architecture introduced in the previous section.

A. Baseline Cluster

For baseline mapping, we start with a greedy clustering program in the spirit of Toronto's vpack/t-vpack [27]. This greedy mapping attempts to minimize the number of clusters. Its key challenge is selecting a set of LUTs that will meet the limited number of inputs to the cluster. Consequently, the greedy packer creates clusters by successively selecting one of the unpacked LUTs that least increases the total inputs into the cluster. Once a cluster is full with four LUTs or there is no unpacked LUT that can be packed into the cluster and meet the input limit, the cluster is considered complete and a new cluster is started. This greedy packer is a little less aggressive than t-vpack, achieving cluster counts that are within 7% of those produced by t-vpack.

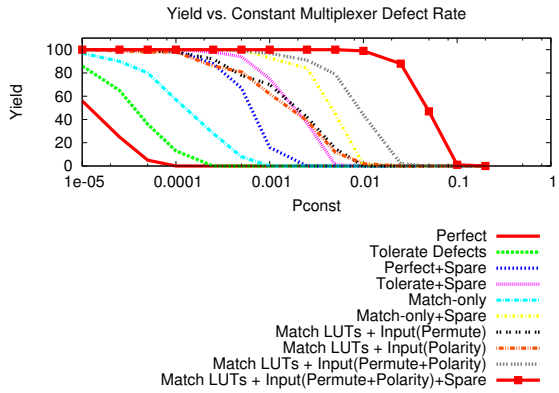


Fig. 5. Comparison of Criteria and Transform Cases for Greedy (Defect-Unaware) Clustering for `clma`

B. In-Clusters Repair

The first set of repairs we consider are all based on local remapping within the cluster. We use the baseline greedy packer that has no specific concern for defects (Sec. VI-A). Local remapping within the cluster is potentially fast since clusters are small, meaning there are few options to explore, and each cluster can be repaired independently. These repairs are all linear in the number of clusters on the chip. Some are super-linear in the number of LUT inputs or the number of LUTs in a cluster.

1) Substitute Spare LUT: For comparison to prior work on full LUT sparing, we consider the case where the only transform is the substitution of the single spare LUT in the cluster with a defective LUT that has been assigned to a logical LUT. That is, we map each LUT in the packed cluster to a specific corresponding LUT in the physical cluster. If the physical LUT is defective, we try to remap the logical LUT to the single spare in the physical cluster. We show two cases in Fig. 5. The "Perfect+Spare" case demands that we only use perfect LUTs, while the "Tolerate+Spare" case allows a LUT to use a partially defective physical LUT if the LUT function can tolerate its constant multiplexer defects. These curves illustrates the benefit of using partially defective LUTs in the most primitive manner. Using the "Tolerate+Spare" case, with one spare, we can achieve over 90% yield for defect rates up to 0.05%, while the "Perfect+Spare" can only exceed 90% yield at 0.01%.

2) Match LUT in Cluster: Instead of simply performing a direct mapping between LUT functions and physical LUTs, we compute a partially defective LUT mapping (Sec. III-E) between the LUT functions in a cluster and the physical LUTs.

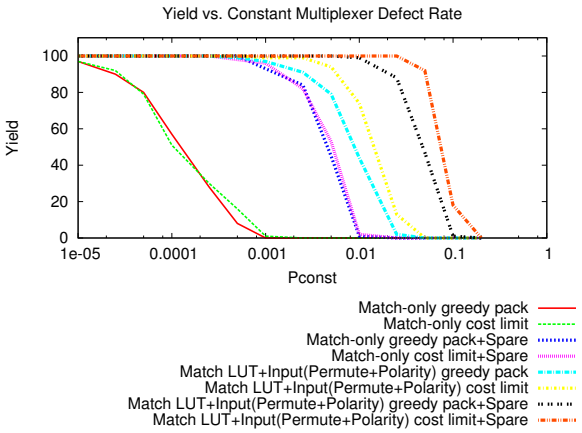


Fig. 6. Impact of Defect-Aware Clustering on `clma`

This can be done efficiently by using a bipartite matching algorithm such as Hopcroft-Karp [28]. Since our illustration architecture only has four LUT functions and four or five physical LUTs, we generate all possible function to physical LUT mappings and check if the mapping tolerates the specific set of constant multiplexer defects. The "Match-only" curve in Fig. 5 shows that this has some yield increase even without a spare, and the "Match-only+Spare" curve shows how the full matching allows more defect tolerance than simple sparing, exceeding 90% yield up to 0.1% defect rates.

3) Input Optimization: We can further considering permuting the LUT inputs, selecting LUT input polarity, or perhaps both in order to move tolerable constant multiplexers to align with physical constant multiplexer failures. The curves annotated with "Input(Polarity)", "Input(Permute)", and "Input(Permute+Polarity)" show the impact of adding these transformations during matching. In practice the polarity optimization is easy if it can be coordinated between all the consumers of a LUT—we just invert the programmed LUT function for that LUT. The exploration here makes the more generous assumption that each LUT input can be independently inverted. We show this range of input optimization in part because, once we add constraints due to depopulated inputs and consistent net polarity, the reality is likely to be somewhere between the extremes of no input optimizations and full input optimization.

C. Defect-Aware Clustering for In-Cluster Repair

In this section, we refine the packing algorithm to enhance the likelihood that the design can be mapped using cluster-local transformations.

We noted earlier (Sec. IV) that different LUT functions have different hardness. Clusters that have a large number of hard functions (*e.g.* four XOR4's) will be the least likely to yield. A defect-aware clustering would avoid creating such hard clusters. Rather, it might pair easy functions (*e.g.* AND4) with hard functions. The generalization of this idea is a form of load balancing. We would like to balance the number of tolerable constant multiplexers so that no cluster ends up with an excessively small number. The tolerable multiplexers distributions shown in Tab. II suggests we have room to perform this kind of constant multiplexer tolerance balancing. To do this, we make two changes to the greedy mapping: (1) we break ties among LUT functions with equal I/O count by selecting one of the LUT functions that maximizes the number of tolerable multiplexers in the cluster, and (2) we set a limit on the number of non-tolerable constant multiplexers in the cluster, rejecting a candidate LUT for clustering if it drops the number of tolerable constant multiplexers in the cluster too low. Furthermore, we try to choose as large a limit as possible while still allowing the design to be packed into a targeted number of clusters (See Alg. 1).

Algorithm 1 Defect-Aware Packing

$mintol \leftarrow CLUSTER\_LUTS \times (2^k - 1)$
loop
  $clusters \leftarrow emptyClusterSet$
  $LUTs\_to\_pack \leftarrow all\_LUTs$
  while $not(empty(LUTs\_to\_pack))$ **and** $clusters.size < MAX\_CLUSTERS$ **do**
    $cluster \leftarrow clusters.newCluster()$
    $cluster.add(LUTS\_to\_pack.getNext())$
    repeat
      $best \leftarrow none$
      for $l \in LUTS\_to\_pack$ **do**
        $tmp \leftarrow cluster.extend(l)$
        if $tol\_mux(tmp) \geq mintol$ **and** $cluster\_io(tmp) < CLUSTER\_INPUTS$ **then**
          if $cluster\_io(tmp) < cluster\_io(best)$ **then**
            $best \leftarrow tmp$
          else if $cluster\_io(tmp) = cluster\_io(best)$ **and** $tol\_mux(tmp) > tol\_mux(best)$ **then**
            $best \leftarrow tmp$
          end if
        end if
      end for
      $cluster.add(best)$
      $LUTS\_to\_pack.remove(best)$
    until $cluster.size = CLUSTER\_LUTS$ **or** $best = none$
  end while
  if $empty(LUTs\_to\_pack)$ **then**
    return $clusters$
  end if
  $mintol = mintol - 1$
end loop

Fig. 6 shows the improvements offered by using this defect-aware clustering as compared to the simpler greedy clustering algorithm. Note that the defect-aware packer has its biggest impact in raising the yield to 100% for some defect rates. It is the unfortunate cases where a relatively non-tolerant cluster happens to be mapped to a particularly defective physical cluster that causes this yield loss. The tolerance balancing makes it more likely all clusters will tolerate the most defective physical cluster on the chip. With a spare, this allows us to achieve over 90% yield for defect rates up to 5%.

D. Defect-Aware Placement

Abandoning the benefits of cluster-local repairs, we can perform full, defect-aware placement to tolerate even higher

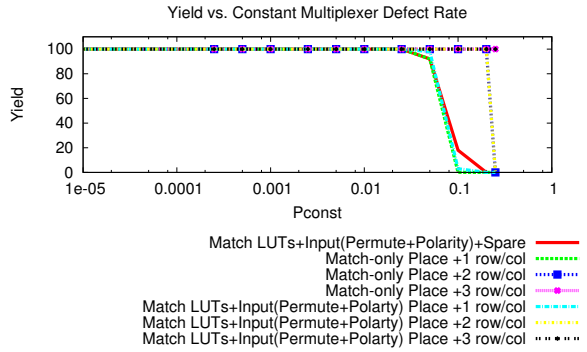


Fig. 7. Impact of Defect-Aware Placement on `clma`

defect rates. With full placement, instead of forcing a LUT to find a match within the cluster, we open up the possibility for it to find a tolerable physical LUT anywhere on the chip. This creates a larger set of match candidates making it much more likely we can tolerate a set of defects. Yield will no longer be limited by a few particularly defective clusters.

Our defect-aware placement begins with an initial assignment of LUT functions to specific clusters that is similar to clustering. In the placement case we know the exact defect patterns in each cluster, so we can make sure we never assign a set of LUTs to a cluster that cannot tolerate its defects. Once we succeed with an initial placement, we perform incremental improvement to reduce wirelength using simulated annealing. During the simulated anneal we only swap LUTs if the clusters continue to tolerate defects after the swap.

Since we know the exact location of the defects, we can avoid defects as long as there are enough physical LUTs from which to select. We characterize the defect rate achievable for 0–3 extra row and column pairs (Fig. 7) compared to the defect-aware packer with a spare LUT. The initial packing for the greedy clustering case required an extra row and column, so we show that along with two and three extra rows and columns. This shows that defect-aware placement can reasonably tolerate 25% constant multiplexer failures. We did not test defect rates above 25%. The probability of a LUT being perfect when each of 14 multiplexers fail with probability 25% is $(0.75)^{14}$ =1.8%, or, equivalently, this represents a perfect LUT failure rate of over 98%.

The full placement results also serve to bracket the tolerable defect rates with simpler schemes. Between only allowing matching within a cluster and allowing matching across the entire chip there are a range of intermediate approaches. For example, we might restrict matching to adjacent clusters in the array, or, more generally, clusters within a specified Manhattan distance. These limited techinques could be used to handle lower defect rates while limiting the per component runtime needed to perform the placement.

VII. Results

Tab. III shows the result of mapping the Toronto20 benchmarks using a selection of the mapping strategies detailed in the previous section. For compactness, we simply show the highest percentage of defects that can achieve over 90% yield for each mapping scenario. This shows that the illustrative trends for `clma` presented in the previous section are typical of the entire benchmark suite.

Demanding perfect LUTs, we cannot tolerate even 0.01% multiplexer failure. This underscores why LUTs could be a limit to running minimum size devices below about 400mV even in 22 nm technology (Fig. 2). With defect-aware packing, input transforms, and tolerance of constant multiplexer failures, we can accommodate around 1% failures without adding spares—enough to allow even minimum-sized devices to run down to 300mV. With spares or modestly sized devices, these techniques should be sufficient to guarantee that LUTs are not the limitation for operation down to 200mV—the operating point achieved for interconnect in [1].

VIII. Future Work

In this work, we have treated the logic mapping as given. *LUT covering* could also be used to control the generation of hard LUT functions, like XOR4, by, for example, rejecting any LUT covers with too few tolerable constant multiplexers. A full solution and characterization will need to *combine interconnect defect tolerance with LUT defect tolerance*, including dealing with the interaction between potentially depopulated C-box and intra-LUT crossbars and LUT input permutation.

We have specifically demonstrated the techniques on 4-LUTs. The strategy here extends to any sized LUTs and will be even more important with larger LUTs. We have not dealt with the more complex logic structures, such as Adaptive Logic Modules [6] in Altera's Stratix 2, or with carry chains present in all commercial FPGA designs. These may restrict the opportunities for LUT matching and present a need for additional techniques for local remapping to tolerate LUT multiplexer failures. Furthermore, the focus of this paper was on demonstrating and characterizing techniques rather than architectural optimization. Characterizing how defects and the use of these techniques impact the best choice of *architectural parameters* (*e.g.* LUT size, cluster size, population, segmentation) remains future work.

IX. Conclusions

Most LUT functions mapped to FPGAs can tolerate a few constant multiplexer failures in the physical LUT. Leveraging a few transformation we can exploit this fact to increase the chance that a LUT function can be successfully mapped to a partially defective physical LUT. We show that this both allows us to handle a high rate of constant multiplexer failures through local, within-cluster sparing without adding spares, around 1% with defect-aware clustering, and that this also increases the tolerable failure rate with spares to around 5%. We also demonstrate that full defect-aware placement can tolerate constant multiplexer failure rates in the 10–25% range.

X. Acknowledgments

This research was funded in part by National Science Foundation grant CCF-0904577 and DARPA/CMO contract HR0011-13-C-0005. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not reflect the official policy or position of the National Science Foundation, Department of Defense, or the U.S. Government.

TABLE III. TOLERABLE PERCENTAGE OF CONSTANT MULTIPLEXER DEFECTS FOR 90% YIELD

Design	LUTs	Greedy Pack							Defect-Aware Pack					Defect-Aware Place			
		Clusters	Perf +Spare	Tol	Tol +Spare	Match	Match +Input	Match +Input +Spare	Clusters	Δ%	Match	Match +Input	Match +Input +Spare	Match Only extra row/cols +0	+1	+2	+3
alu4	1102	312	0.05	0.001	0.1	0.01	1	5	323	3	0.01	1	5	0	10	25	25
apex2	1303	361	0.05	0.001	0.1	0.005	1	2.5	393	8	0.005	1	5	2.5	25	25	25
apex4	1045	284	0.05	0.0025	0.25	0.01	1	5	292	2	0.005	1	5	0	25	25	25
bigkey	1327	337	0.025	0.001	0.05	0.001	0.5	2.5	577	71	0.1	5	10	25	25	25	25
clma	4008	1064	0.01	0	0.05	0.0025	0.25	1	1079	1	0.0025	0.5	5	0	5	20	25
des	1232	318	0.025	0.001	0.1	0.0025	0.025	0.5	841	164	1	1	2.5	25	25	25	25
diffeq	912	233	0.05	0.001	0.1	0.0025	0.05	1	263	12	0.005	1	2.5	10	20	20	25
dsip	1108	282	0.025	0.001	0.1	0.001	0.5	2.5	670	137	0.5	10	10	25	25	25	25
elliptic	2043	531	0.025	0	0.05	0.001	0.025	0.5	561	5	0.001	0.25	1	0	0.5	10	10
ex1010	3505	953	0.025	0.001	0.1	0.0025	0.5	5	954	0	0.005	0.5	5	0	0	0	20
ex5p	756	204	0.05	0.0025	0.1	0.01	1	5	214	4	0.01	1	5	0	20	25	25
frisc	2323	594	0.025	0	0.05	0.001	0.025	0.5	662	11	0.0025	0.5	2.5	0	10	10	20
misex3	1044	295	0.05	0.001	0.25	0.01	0.5	2.5	313	6	0.01	1	5	0	25	25	25
pdc	3004	828	0.025	0	0.05	0.0025	0.05	1	835	0	0.0025	0.25	1	0	10	20	25
s298	879	243	0.05	0.0025	0.25	0.01	1	5	254	4	0.01	1	5	0	10	25	25
s38417	3401	873	0.025	0	0.05	0.001	0.01	0.5	939	7	0.0025	0.25	1	10	20	25	25
s38584.1	3909	990	0.025	0	0.05	0.001	0.01	1	1045	5	0.0025	0.5	1	10	20	25	25
seq	1161	327	0.05	0.001	0.1	0.005	1	5	342	4	0.01	1	5	5	25	25	25
spla	2495	693	0.025	0	0.05	0.0025	0.1	1	709	2	0.005	0.5	2.5	0	0	10	25
tseng	778	199	0.05	0.0025	0.1	0.005	0.05	1	222	11	0.01	1	2.5	0	10	20	20

Based on discrete mappings at P_{const}={0.00001, 0.000025, 0.00005, 0.0001, 0.00025, 0.0005, 0.001 0.0025, 0.005, 0.010, 0.025, 0.05, 0.10, 0.20, 0.25}

REFERENCES

[1] N. Mehta, R. Rubin, and A. DeHon, "Limit Study of Energy & Delay Benefits of Component-Specific Routing," in *FPGA*, 2012, pp. 97–106.

[2] V. Lakamraju and R. Tessier, "Tolerating operational faults in cluster-based FPGAs," in *FPGA*, 2000, pp. 187–194.

[3] J. Rose, R. Francis, D. Lewis, and P. Chow, "Architecture of field-programmable gate arrays: The effect of logic block functionality on area efficiency," *IEEE Journal of Solid-State Circuits*, vol. 25, no. 5, pp. 1217–1225, October 1990.

[4] K. Poon, S. Wilton, and A. Yan, "A detailed power model for field-programmable gate arrays," *ACM Tr. Des. Auto. of Elec. Sys.*, vol. 10, pp. 279–302, 2005.

[5] J. Luu, I. Kuon, P. Jamieson, T. Campbell, A. Ye, W. M. Fang, and J. Rose, "VPR 5.0: FPGA cad and architecture exploration tools with single-driver routing, heterogeneity and process scaling," in *FPGA*, 2009, pp. 133–142.

[6] D. Lewis, E. Ahmed, G. Baeckler, V. Betz, M. Bourgeault, D. Cashman, D. Galloway, M. Hutton, C. Lane, A. Lee, P. Leventis, S. Marquardt, C. McClintock, K. Padalia, B. Pedersen, G. Powell, B. Ratchev, S. Reddy, J. Schleicher, K. Stevens, R. Yuan, R. Cliff, and J. Rose, "The Stratix-II logic and routing architecture," in *FPGA*, 2005, pp. 14–20.

[7] V. Betz and J. Rose, "Cluster-based logic blocks for FPGAs: Area-efficiency vs. input sharing and size," in *CICC*, May 1997, pp. 551–554.

[8] W. B. Culbertson, R. Amerson, R. Carter, P. Kuekes, and G. Snider, "Defect tolerance on the TERAMAC custom computer," in *FCCM*, April 1997, pp. 116–123.

[9] R. Amerson, R. Carter, W. B. Culbertson, P. Kuekes, and G. Snider, "Plasma: An FPGA for million gate systems," in *FPGA*, February 1996, pp. 10–16.

[10] J. Lach, W. H. Mangione-Smith, and M. Potkonjak, "Low overhead fault-tolerant FPGA systems," *IEEE Trans. VLSI Syst.*, vol. 6, no. 2, pp. 212–221, June 1998.

[11] G. Lemieux, E. Lee, M. Tom, and A. Yu, "Directional and single-driver wires in FPGA interconnect," in *ICFPT*, December 2004, pp. 41–48.

[12] N. Mehta, "An ultra-low energy, variation tolerant FPGA architecture using component-specific mapping," Ph.D. dissertation, California Institute of Technology, 2013. [Online]. Available: http://resolver.caltech.edu/CaltechTHESIS:10072012-230900231

[13] A. Asenov, "Random dopant induced threshold voltage lowering and fluctuations in sub-0.1 μm MOSFET's: A 3-D "atomistic" simulation study," *IEEE Trans. Electron Devices*, vol. 45, no. 12, pp. 2505–2513, December 1998.

[14] A. Asenov, S. Kaya, and A. R. Brown, "Intrinsic parameter fluctuations in decananometer MOSFETs introduced by gate line edge roughness," *IEEE Trans. Electron Devices*, vol. 50, no. 5, pp. 1254–1260, May 2003.

[15] A. Asenov, "Intrinsic threshold voltage fluctuations in decanano MOSFETs due to local oxide thickness variation," *IEEE Trans. Electron Devices*, vol. 49, no. 1, pp. 112–119, January 2002.

[16] V. A. Sverdlov, T. J. Walls, and K. K. Likharev, "Nanoscale silicon MOSFETs: A theoretical study," *IEEE Trans. Electron Devices*, vol. 50, no. 9, pp. 1926–1933, September 2003.

[17] K. Bernstein, D. J. Frank, A. E. Gattiker, W. Haensch, B. L. Ji, S. R. Nassif, E. J. Nowak, D. J. Pearson, and N. J. Rohrer, "High-performance CMOS variability in the 65-nm regime and beyond," *IBM J. Res. and Dev.*, vol. 50, no. 4/5, pp. 433–449, July/September 2006.

[18] "International technology roadmap for semiconductors," <http://www.itrs.net/Links/2011ITRS/Home2011.htm>, 2011.

[19] S. Hanson, B. Zhai, K. Bernstein, D. T. Blaauw, A. Bryant, L. Chang, K. K. Das, W. Haensch, E. J. Nowak, and D. Sylvester, "Ultralow-voltage, minimum-energy CMOS," *IBM J. Res. and Dev.*, vol. 50, no. 4–5, pp. 469–490, 2006.

[20] W. Zhao and Y. Cao, "New generation of predictive technology model for sub-45 nm early design exploration," *IEEE Trans. Electron Dev.*, vol. 53, no. 11, pp. 2816–2823, 2006.

[21] S. Mukhopadhyay, H. Mahmoodi, and K. Roy, "Modeling of failure probability and statistical design of sram array for yield enhancement in nanoscaled cmos," *IEEE Trans. Computer-Aided Design*, vol. 24, no. 12, pp. 1859–1880, December 2005.

[22] V. Betz and J. Rose, "FPGA Place-and-Route Challenge," <http://www.eecg.toronto.edu/~vaughn/challenge/challenge.html>, 1999.

[23] A. Mishchenko, R. Brayton, J.-H. R. Jiang, and S. Jang, "Scalable don't-care-based logic optimization and resynthesis," *ACM Tr. Reconfig. Tech. and Sys.*, vol. 4, no. 4, pp. 34:1–34:23, Dec. 2011.

[24] J. Rose *et al.*, "VPR and T-VPack: Versatile Packing, Placement and Routing for FPGAs," <http://www.eecg.utoronto.ca/vpr/>, 2009.

[25] K. Fujiyoshi, Y. Kajitani, and H. Niitsu, "Design of minimum and uniform bipartites for optimum connection blocks of FPGA," *IEEE Trans. Computer-Aided Design*, vol. 16, no. 11, pp. 1377–1383, November 1997.

[26] G. Lemieux and D. Lewis, "Using sparse crossbars within LUT clusters," in *FPGA*, 2001, pp. 59–68.

[27] A. Marquardt, V. Betz, and J. Rose, "Using Cluster-Based Logic Blocks and Timing-Driven Packing to Improve FPGA Speed and Density," in *FPGA*, February 1999.

[28] J. E. Hopcroft and R. M. Karp, "An $n^{2.5}$ algorithm for maximum matching in bipartite graphs," *SIAM Journal on Computing*, vol. 2, no. 4, pp. 225–231, 1973.