Scalable Defect Mapping and Configuration of Memory-Based Nanofabrics

Chen He[1], Margarida F. Jacome, and Gustavo de Veciana
Department of Electrical and Computer Engineering
University of Texas at Austin
{che,jacome,gustavo}@ece.utexas.edu
[1]Also with Freescale Semiconductor, Inc.

***Abstract*— Producing reliable nanosystems requires effectively addressing the high defect densities projected for nanotechnologies. Defect avoidance methodologies based on reconfiguration offer a promising solution to achieve defect tolerance. The idea is to start by obtaining a defect map of the target nanofabric, and then configure the desired functionality 'around' its defective components. In this paper, we argue for the suitability of memory-based computing nanofabrics, address the level of granularity at which defect mapping and configuration should be performed on such fabrics, and discuss the role of hierarchy towards controlling complexity. We then propose a novel group testing method to enable self-testing and self-configuration for appropriately architected memory-based nanofabrics. The proposed testing method is scalable and simple, in that it enables the entire fabric to be tested and configured using a relatively small number of easily configurable triple-module-redundancy (TMR) test tiles executing concurrently on different regions of the target nanofabric. Our experimental results demonstrate the effectiveness of the proposed method for a representative set of benchmark kernels.**

I. Introduction

Significant advances have been made towards realizing the promise of emerging nanotechnologies, including devising novel nanoelectronic devices and successfully assembling them into logic gates and memory arrays [1], [2], [3]. These striking successes are at the basis of current projections placing the ability to manufacture large-scale computation nanofabrics in a 10-15 year window [4].

Yet, if this ambitious timeline is to be met, it is of paramount importance to devise novel system architectures and design paradigms that can effectively address the tremendous reliability and scalability challenges intrinsic to nanotechnologies [5]. Specifically, the aforementioned nanofabrics are expected to exhibit a high density of hard faults or *defects* as well as a high susceptibility to soft or *transient faults* [4], [6], [7]. As technologies mature, one may expect some improvements in this arena, yet the fundamental reliability 'problem' will persist. Indeed, decreasing the size of structures increases the ratio of surface area to volume and, thus, naturally occurring imperfections on materials' surfaces or boundaries will more severely compromise our ability to ensure that a fabric's interconnects and devices will be appropriately formed. Furthermore, at the reduced scales associated with nanoelectronic devices, the energy required to cause faults decreases and, thus, the flux density of particles able to cause errors quickly increases. These observations point to a reliability problem which is intrinsic to nanoscale regimes and is thus here to stay. Moreover, in order to leverage the unprecedented densities afforded by nanotechnologies (on the order of 10^{12} devices per cm^2), the scalability of computing architectures (in terms of performance/speed and power) and the scalability of associated design and test methodologies (in terms of complexity and practicality) will be critical.

Early on, the creators of TERAMAC [8] identified the possibility of utilizing reconfiguration to achieve tolerance to hard defects in systems targeted at emerging nanotechnologies [6], since the underlying assumptions of such an approach's – sufficient communication bandwidth and abundant redundant resources – would still be valid at the nanoscale level of integration. Furthermore, nanostructures well suited to building reconfigurable computational fabrics have already been demonstrated, see e.g., [1], [9], [10], making reconfiguration-based approaches to defect avoidance promising, see also [6]. Indeed, even if mapping defects and then configuring the target functionality around such defects, on a per-chip basis, is a challenging task, the alternative – using brute force redundancy (temporal and/or spatial) to ensure a sufficiently high probability of correct chip operation – would likely be very inefficient in terms of power/energy, incur a substantial performance/delay overhead, and would ultimately be limited by the reliability of the required arbitration circuits, see e.g., [11], [12]. Coding does not provide a reasonable alternative either. In [13], for example, Von Neuman's multiplexing scheme is used to achieve soft fault tolerance – specifically, the paper proposes a multi-stage multiplexing scheme with restoration to reduce redundancy overheads. However, the authors admit that the resulting overheads are still excessive, and suggests that a defect avoidance technique based on reconfiguration would be more effective in terms of handling defects [14].

In order to establish the novelty and relevance of our paper, we start by making an important initial observation: the scalability and practicality of any reconfiguration-based defect-avoidance approach is strongly predicated on the type and granularity of the primitive programmable elements adopted for the target nanofabric, as well as on the fabric's overall organization. Specifically, a primary concern should be to architect the nanofabric so as to enable each chip to execute its own standard self-testing procedure, followed by a self-configuration step, requiring minimal off-line processing (ideally none). A seminal defect avoidance approach for nanotechnologies was proposed in [9], [15], where the target reconfigurable nanofabric is a large regular grid of nanoblocks, each of which capable of implementing a small logic block, with only a few gates. Defects on the grid's nanoblocks are first mapped on each chip, using a *two-phase group testing* technique (see details in Section IV). Then, for each chip and associated defect map, a feasible configuration is synthesized (off-line) realizing the application functionality 'around' defective nanoblocks, and finally the corresponding chip is configured accordingly. Unfortunately, this approach faces substantial limitations and scalability challenges. In particular, it requires *off-line processing* specific to each *individual* chip, which is a potential show stopper to achieving cheap and fast mass-production. A central problem is the small level of granularity at which defect mapping and configuration are performed, making it very challenging to implement and orchestrate the full process entirely within the chip.

Ideally, one would like to use the processing power of the fabric

itself to perform the defect mapping and configuration tasks, in a simple way. The abstract nanofabric architecture we proposed in [11], [12] shows definitive promise in that direction, that is, in terms of enabling scalable self-testing and self-configuration methods for defect prone nanotechnologies. Yet, in [11], [12] we did not provide discussion, or practical insight, on how the fabric's primitive elements could be (tentatively) implemented using programmable nanostructures described in the current literature (e.g., [1], [2]). Since the adopted primitive elements are much coarser than those used in previous approaches, e.g., [15], [10], demonstrating their practicality is very critical. Moreover, in [11], [12] we did not discuss on how to actually implement the suggested tile-based self-testing method.

Accordingly, in this paper we revisit the abstract nanofabric architecture proposed in [11] and address a number of important open issues. Specifically, we first argue for the promise of 'memory-based computing' in the context of emerging nanotechnologies (see also [6]). We then revisit the reconfigurable memory-based fabric architecture, assess its adequacy in terms of supporting scalable defect avoidance methods, and discuss possible implementations for some of its key elements. Furthermore, we consider the self-testing idea suggested in [11] and propose a detailed group testing methodology and a concrete potential design of the required supporting circuitry, relying on programmable nanostructures described in the current literature. Finally, we discuss critical trade-offs between testing performance and complexity associated with the proposed method.

The paper is organized as follows. In Sections 2 and 3 we argue on the suitability of memory-based computing nanofabrics, address the level of granularity at which defect mapping should be performed on such fabrics, and discuss the role of hierarchy towards controlling complexity. We then review the nanofabric architecture adopted in the paper and provide potential implementation solutions for some of its key elements. In Section 4 we review previous relevant work in group testing. In Section 5 we propose a detailed group testing methodology for defect mapping on our target nanofabric. Possible designs of required supporting circuitry are presented and analyzed in Section 6. Experimental results showing the effectiveness of the proposed method are given in Section 7. Finally, Section 8 concludes the paper.

II. A Case for 'Coarse-Grained' Memory-Based Computing Nanofabrics

Computers built solely of wires, switches and memory-based look-up tables (LUTs), i.e., requiring no traditional logic gates (or very few of these), are very appealing in the context of nanotechnologies [6]. The appeal of such memory-only computers lies in the fact that, one can rely solely on simple, highly regular, and ultra dense fabrics comprised of crossbar structures, to build powerful substrates capable of performing arbitrarily complex computations.

For concreteness, Fig.1 shows a possible nanowire crossbar-based memory structure, denoted the Harvard-CalTech nanomemory [16], [17]. This particular architecture contains: (1) a crossbar nanowire memory array comprised of nonvolatile nanoscale cross-switches – possible realizations for the latter include suspended nanotube switches [2], crossed nanowire diodes [16], and rotaxanes-based molecular switches [1]; and (2) a row decoder and a column decoder – possible implementations of such decoders may rely on crossed semiconductor nanowire (cNW) field-effect transistor (FET) arrays [18], or FET arrays formed by modulation-doped nanowires with top-gated microwires [17]. Of course, one can also implement conventional logic gates using such programmable nanowire crossbar structures. Indeed, several architectures comprised of functional

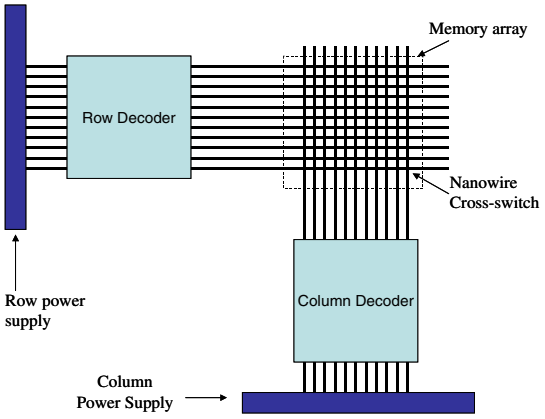

Fig. 1. Nanowire crossbar-based memory structure.

nanoarrays implementing programmable logic array (PLA) planes (e.g., OR and NOR) have been proposed, see e.g., [9], [10]. While such architectures can provide universal logic functionality, i.e., their logic planes can be configured to compute any logic function, performing defect mapping and *self-configuration* at such a fine level of granularity (e.g., OR and NOR elements) can be quite challenging, particularly when ultra dense nanochips are considered.[1]

In order to enable each nanochip to orchestrate its own self-testing and self-configuration procedures (requiring minimum or no off-line processing), the level of granularity at which defect mapping is done should be 'coarser'. To this end, in [11], we proposed a fabric comprised of a grid of processing elements (PEs) capable of executing 8-bit arithmetic and logic operations, and showed that one could use the processing power of such a fabric to map defects – namely, PEs may be efficiently used to test their neighbor PEs and fabric connectivity. Before considering in more detail our self testing ideas suggested in [11], we address key implementation issues and practicality concerns pertaining the PEs themselves (not considered in [11]).

A solution that we find very appealing would be to implement each arithmetic and logic operation supported by a PE (e.g., addition, subtraction, etc.) directly on one or more dedicated LUTs. The underlying idea is to favor simplicity and regularity in the realization and *testing* of PEs (see Section V), rather than trying to use 'smaller' (more traditional) gate-level implementations of a PE's functional units, since those would be more complex to successfully place and route, as well as test – recall that our ultimate goal is to minimize the need for *off-line processing* specific to *individual* chips, since this may be an impediment to low-cost and fast mass-production. Although the LUTs implementing such operations will be relatively large when compared to those of traditional FPGAs (see Section VI), they can be realized in a very compact and regular way, exploiting the favorable characteristics of the array-based nanomemory architectures alluded to above. Obviously, if excessive size becomes an issue, the lookup tables themselves may be built out of banks of several such crossbar structures, see e.g., [16], each storing results for a specific range of operands. Or one may reduce the overall memory size by building a 'cascaded' LUT implementation of the operation – see examples in Section 6.1.

Naturally, the set of lookup tables comprising each PE needs to

[1]Note that memory-only computing fabrics would suffer from a similar problem, if built out of 'fine grained' primitive programmable elements, say, 64 bit LUTs.

be programmed with the various supported operations, using again a process as general (i.e., non-chip specific) as possible. A possible direction towards achieving the latter would be to use a set of *auxiliary memories* placed at the periphery of the nanochip, to drive the programming of the PEs' lookup tables. Specifically, a stochastic scheme based on the principles described in [17] could be used to derive a deterministic mapping between the actual auxiliary memory addresses and the corresponding logic LUT entries (for a particular operation). That mapping could then be used to appropriately program the auxiliary memory. Ideally, the values stored on these auxiliary memories could then be used to directly program the actual PEs' lookup tables, without the need to leave the nanoscale boundary. Details on this are beyond the scope of this paper and will be reported elsewhere.

Note finally that some level of internal redundancy may be needed in order to enable such 'coarse' PEs to be treated as primitive elements in our methodology, that is, to ensure a likelihood of failure within acceptable limits (say, no more that 20%, as suggested in [7]). Such local redundancy may be incorporated into the PEs, for example, by duplicating look-up tables (or table partitions), and making each individual copy selectable via a 'higher order' bit. Specifically, during the testing of a PE (as discussed in Section VI), for each logic table entry, one would first test the value stored in one array (say, array '0') and then the value stored in the other (array '1'), and then store a bit indicating the specific copy to be considered for that particular entry. This scheme can be trivially extended to incorporate more redundancy, if need be.

III. Defect Mapping and Configuration on Suitably Architected Memory-Based Nanofabrics

Our target nanofabric is architected as proposed in [11], shown in Fig. 2. The basic configuration unit of the nanofabric, called a *region*, is a grid of processing elements (PEs) and switching elements (SEs). Fig.2 shows one such region – a 4×4 grid consisting of 8 PEs and 8 SEs. Recall that, as discussed above, our PEs are quite simple – they perform standard 8-bit arithmetic/logic operations, and are comprised of a small set of look-up tables (LUTs), implementing the various operations, and simple control logic. We will show that the use of such small PEs as the atomic fabric elements enables a simple and yet effective testing methodology for defect mapping.

Each region of the nanofabric can be configured to execute a small behavioral segment, called a *basic flow*. A set of representative basic flows is shown in Fig.3, where each node represents an arithmetic/logic operation to be executed by a PE, and edges represent data transfers between operations, performed through an SE. Naturally, the larger the resource redundancy in a region, the larger the number of alternative configurations for its associated basic flow, and thus the higher the probability of successfully instantiating it on that region. Fig.4, for example, shows 4 out of 191 possible configurations in which a basic flow $ft3$ can be mapped to a 4×4 region. Note that whether a configuration is feasible depends on the defect distributions in the region.

A critical observation we made in [11] is that by architecting the nanofabric in terms of such regions, one decomposes the nanosystem's complex defect mapping and configuration problem into a set of quasi-independent subproblems, each with the scope of a single region and basic flow. Specifically, one can map defects in each region, and then configure individual (limited size) basic flows around

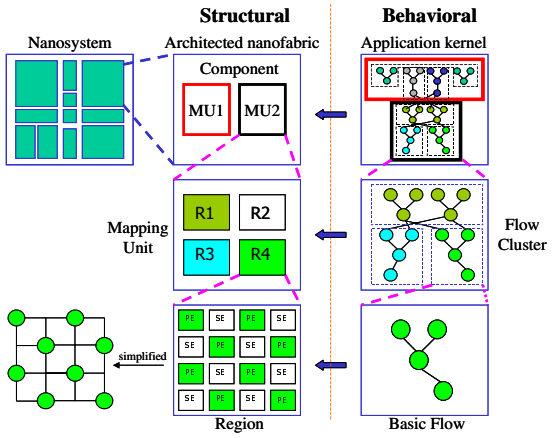

Fig. 2. Hierarchy of design abstractions.

them, making the approach scalable [2]. Indeed, if the number of alternative configurations per region is not excessively high, a simple table-look-up algorithm can be used to find a feasible configuration for a basic flow.

To achieve a sufficiently high probability of successful configuration in a scalable way, we have introduced an additional hierarchical/aggregation level in the fabric architecture, denoted *mapping unit* (MU) [11]. Namely, one may allow a given basic flow to be instantiated in one of n regions in an MU rather than a single (possibly large) one. Or, more generally, one may allow m basic flows to be instantiated in an MU containing n regions, where m is less than or equal to n. Note that the notion of *mapping unit* creates a second level of redundancy, while retaining the original simplicity of the region-based defect mapping and configuration. In [11] we discussed the complexity of routing inside a mapping unit can be effectively controlled, by limiting the maximum number of regions per mapping unit.

In summary, as shown in Fig.2, when designing a reconfigurable nanofabric to implement the components of a nanosystem, the behavior of each such component is first decomposed into a number of basic flows – we may think of these as the 'instruction selection' phase of the synthesis/compilation process. Then small sets of such basic flows are assigned to the component's mapping units, each with as many regions as needed to achieve the target probability of successful configuration.

In [11], we discussed the delay vs. yield trade-offs exposed by this abstract hierarchical organization, and the implications of such trade-offs on synthesis algorithms. In contrast, in this paper we focus on critical realization and practicality issues. In particular, we propose a detailed defect mapping methodology applicable to such architected nanofabrics, including the design of support circuitry, and discuss critical complementary trade-offs between testing coverage and complexity.

IV. Previous Related Work on Group Testing

Suppose we have n items, d of which are defective, and we are interested in finding the best way to identify the d defective items. As the name suggests, in group testing [20] one picks subsets of the n items and determines whether there is a defective item in each such subset or group, and then collectively determine the defects in all n

[2]A similar idea of using alternate configurations for sub-components, i.e. tiles, has been proposed for fault-tolerant FPGAs in [19]. However, no defect/fault detection method was proposed in that paper.

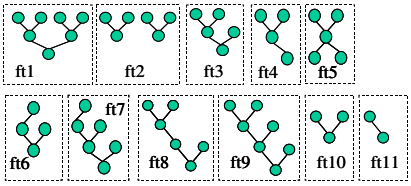

Fig. 3. A representative set of basic flows.

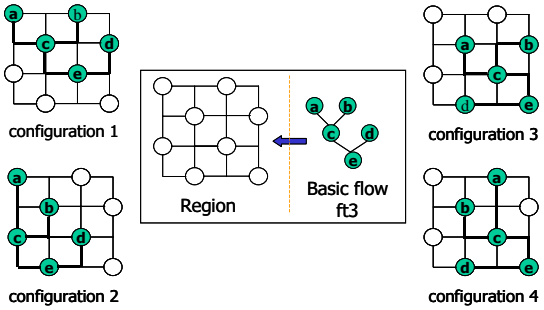

Fig. 4. Map a flow to a region.

items based on the testing results of all groups. In conventional group testing methods, it is assumed that a test group gives a 'positive' if and only if all the items in the group are not defective [20]. Thus, those methods work only when the defect density is not high (see [8]).

To address the challenge posed by the high defect densities projected for nanotechnologies, a *two-phase group testing methodology* was proposed in [15]. During the first 'probability-assignment' phase, one attempts to either get the exact number of defects or a crude estimate for the number of defects e.g., none, some, or many, in the group. Using Bayesian methods, one then analyzes the outputs of all test groups to obtain defect probabilities for individual items. As a result, at the end of this first phase all items are classified into two sets: those with high probability of being good and those with high probability of being defective. During the second 'defect location' phase, conventional group testing is applied, but only to the subset of items likely to be good, with the goal of identifying those which are defective. Unfortunately, this second phase of the proposed group testing methodology requires unlimited connectivity among the nanofabric's components [15]. The CAEN-BIST method proposed in [21] improves over [15], in that it does not require unlimited connectivity. Still, approaches relying on these two methods would ultimately require that an off-line mapping of the target functionality 'around' the identified defects be performed of a per-chip basis, and thus will not scale for large nanosystems.

V. TMR-Based Group Testing Methodology

In this section we define a novel group testing methodology aimed at effectively handling the high defect densities projected for nanotechnologies.

A. Group Testing with TMR Test Tiles

The idea of performing defect mapping on a nanofabric's regions using Triple-Module-Redundancy (TMR) test tiles was first proposed by us in [11]. A TMR test tile (or group) is formed by configuring four PEs, where one plays the role of an *arbiter* for the outputs of the other three – the latter are referred to as the tile's *peer* PEs. These small tiles can be configured systematically, and can usually locate defective PEs and/or connections[3] in the region (see also Section VI).

Since simplicity is of paramount importance to enable self-testing of large nanofabrics, we propose a simple conservative algorithm to obtain a (partial) defect map of a region when applying a set of TMR test tiles to it. Specifically, we start by assuming that all PEs and connections in a region are defective. Then, for each tile in the testing suite, if the tile 'passes' (see details in Section VI) we update the set of PEs and connections that are known to be good, otherwise we do nothing. Clearly, this algorithm is conservative, in that bad PEs or connections are very unlikely to be marked as good [4], yet good PEs/connections may be marked as bad, i.e., false negatives can be generated. Note, however, that the probability of such false negatives can be made very small in our approach. Specifically, since each PE is included and tested in different tiles (see Section 5.3), the redundancy in the testing process makes it quite effective in identifying defects. Note, finally, that this simple algorithm is amenable to a lookup table (LUT) implementation, thus providing a good foundation towards minimizing offline processing.

The performance measure for our flow-oriented defect testing method is *flow coverage*, which is defined as one minus the probability of false negatives, when configuring a basic flow. It has a direct impact on *flow yield*, which is a primary figure of merit for a design, defined as the probability of successfully configuring a flow on a target region. Flow yield is given by the probability of successfully configuring a flow on a region with a perfect defect map minus the probability of a false negative. It thus depends upon the flow characteristics, such as size and connectivity, as well as on the flow coverage of the test method.

B. Considerations on Region Size

As alluded to before, it is critical to contain the complexity of the defect mapping and chip configuration tasks, so that both can be addressed/performed using the processing power and storage capacity of the actual nanochip. The 4×4 grid, shown in Fig.2, realizes a specific trade-off between *configuration capacity*, i.e., the amount of raw redundant capacity provided at the region level, and *complexity*, i.e., the number of TMR test tiles and the number of alternate configurations required to achieve a good flow coverage. For example, adopting a larger region size, say, an 8×8 grid, would increase the amount of configuration capacity at the region level, and thus increase flow yield. However, it would also lead to a tremendous increase on the number of TMR test tiles one can configure on a region – from 58 for a 4×4 region, to 866 for a 8×8 region, as well as an exponential growth on the number of alternate configurations for each basic flow – this is shown in Fig.5. Thus an increase in the region size would lead to excessive memory requirements to support the self-configuration phase. As we pointed out in [11], the configuration capacity (and thus yield) can alternatively be increased by adding more regions to a mapping unit, and/or using smaller basic flows, etc. Thus, keeping the size of the fabric's regions relatively small is a good strategy to control complexity and ensure practicality for our approach. Therefore, in this paper we consider the 4×4 grid region shown in Fig.2.

[3] A connection represents the set of all possible direct paths between two diagonally or vertically or horizontally adjacent PEs through a switching element.

[4] It is assumed that a PE configured as an arbiter which is faulty is very unlikely to generate a false positive, i.e., generate a 'correct' diagnosis message. This can be induced by requiring the arbiter to generate a long signature bit-stream together with the arbitration result.

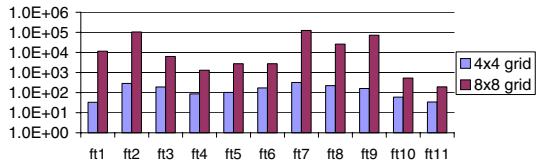

Fig. 5. Number of possible configurations for representative basic flows.

C. TMR Test Tile Selection

As mentioned in the previous section, a total of 58 TMR test tiles can be configured on a 4×4 region. Yet, by using a smaller number of tiles may still achieve good flow coverage, while reducing testing time and required configuration memory. In this section, we address the TMR test tile selection problem, i.e., the selection of a set of tiles that realizes a good trade-off between *flow coverage* and *complexity* in terms of testing time and size of configuration memory. Clearly, the more tiles are used, the better the defect map one obtains, i.e., the better the flow coverage, but the higher the *complexity*.

We used extensive Monte Carlo simulation to estimate the flow coverage for each basic flow achieved by alternative suites of TMR test tiles, assuming a wide range of defect scenarios representing potential defect distributions for future nanotechnologies. Specifically, we assume probabilities of failure for PEs operating as generic processors, denoted P_e, in the range of 1–20%, probabilities of failure for PEs operating as arbiters, denoted P_a, in the range of 0.5-10% [5], and probabilities of failure for connections, denoted P_c, in the range of 0.1–2% [6].

Fig.6 shows the results obtained using different sets of 12 TMR test tiles, for defect regime $(P_e, P_a, P_c) = (10, 5, 1)\%$. *Standard suite* denotes the 12 TMR test tiles shown in Fig.7. *Suite1 – Suite3* denote three alternative, randomly selected suites of 12 TMR test tiles, all of which evenly spread over the PEs and connections in the region, i.e., each PE and connection in the region is covered by 'roughly' the same number of tiles in the suite. The simulation results in Fig.6 suggest that any suite of 12 TMR test tiles which is evenly spread out among all PEs and connections provides a fairly good flow coverage. Similar results were obtained for other defect regimes. These results also show that no single suite of TMR test tiles can provide maximal coverage for *all* basic flows. Thus, in terms of flow coverage, any 12 TMR test tile suite with even spread is a good candidate, yet the so called *standard suite* (see Fig.7) enables one to minimize overall testing cost in time, since its tiles can be configured as pairs and tested in parallel.

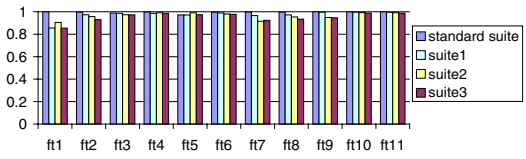

Fig. 6. Flow coverage for basic flows using various suites of 12 TMR test tiles when $(P_e, P_a, P_c) = (10, 5, 1)\%$.

A final interesting empirical observation is that using more than 12 TMR test tiles brings negligible improvement in flow coverage, while using fewer than 12 tiles results in a significant degradation in flow coverage. This is illustrated in Fig.8, where the average flow coverage (computed over all basic flows) obtained using testing suites including 6–14 TMR test tiles is shown. The two extra tiles in the suite of 14 tiles use the 2 internal PEs as arbiters and cover 6 PEs except the 2 corner PEs. The suites with fewer than 12 tiles were obtained by removing tiles from the *standard suite* according to the tile numbering order shown in Fig.7.

[5]See Section VI for a discussion on the relative complexity of general computation vs. arbitration.

[6]One would expect $P_c << P_e$, since our LUT-based PEs are substantially more complex than a switching element/connection.

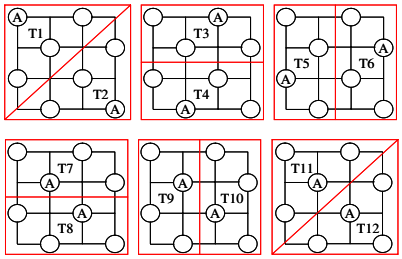

Fig. 7. The standard suite of TMR test tiles. (Arbiter of each tile is labeled 'A'.)

In summary, we have empirically determined that our *standard suite* of 12 TMR test tiles achieves a good trade-off between flow coverage and complexity, and we have thus adopted it in our defect mapping methodology.

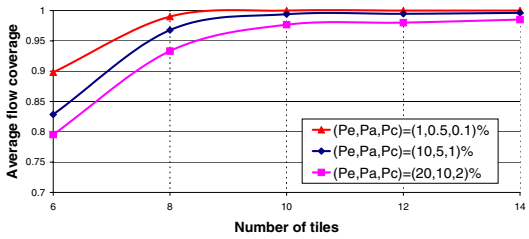

Fig. 8. Average flow coverage under different defect scenarios, and varying number of test tiles .

VI. Supporting TMR-Based Group Testing

In this section we discuss our proposed TMR-based group testing methodology in more detail, presenting its required support circuitry, and showing that such support is commensurate with the small complexity of the processing elements (PEs) – a critical point in terms of demonstrating the effectiveness of our approach. Recall that the standard 8-bit arithmetic/logic operations are implemented in our PEs as nanomemory-based lookup tables (LUTs), with some simple control logic, and possibly redundancy to meet the target reliability at this granularity. For simplicity, the numbers presented below estimate the memory size required by each such operation in terms of the required number of logic LUT entries, assuming no local redundancy. So, we provide a minimum baseline 'cost', but this can then be adjusted to capture the degree of local redundancy required.

To keep the implementation simple, the testing procedure for each TMR testing tile is divided into two steps. The first step tests the connectivity required by the tile (from peer PEs to the TMR tile's arbiter PE, etc.), and the basic control logic for the peer PEs (e.g., operation selection, etc.) During this step, the tile's peer PEs are all configured to perform a common operation, and their results are compared by the arbiter PE for a few inputs. The second step exhaustively tests the correctness of all of the peer PE's operations, i.e., checks if all their LUT entries are defect-free. During the second

step, we scan through all entries of the LUTs (implementing the various arithmetic/logic operations supported by the tile's peer PEs), again using the tile's arbiter to determine the correctness of each result. This second step may be also used to correct errors in the LUTs - see Section 6.2).

A. Step 1 – Testing Control/Connectivity

As mentioned above, to test the basic control and connectivity of the PEs in a TMR tile, the three peer PEs are configured to perform a common operation, say, 8-bit addition, and the arbiter PE is configured to function as a majority-rule based word voter. A short input stream (stored locally to each PE) is then injected into the inputs of the three peer PEs, and their corresponding outputs are sent to the arbiter PE – using its voting/diagnosis circuitry, the latter can signal a 'pass' for this phase of the test or a 'failure'. Since a 'pass' may happen with 0 or 1 faulty PE/connection, the arbiter identifies also the defective PE/connection, when one exists.

Since all PEs in a region will perform the arbitration function in some TMR test tile, our methodology requires incorporating support circuitry in all PEs to perform a majority-rule based 8-bit word voting function. In [22] a word voter design using logic gates is proposed, yet the failing input is not identified by the voter, making it unsuitable to support in our context. We have thus developed an 8-bit word voter using LUTs which is capable of identifying faulty PEs/connections as needed – see Fig.9. Our design has two stages – a vote stage and a diagnosis stage. The first stage is comprised of eight parallel 1-bit majority voters, each accepting inputs from the three peer PEs in the TMR tile – namely, b_{ij} represents the ith bit of data from the jth PE. The output of each 1-bit voter encodes the failing PE for the corresponding bit (if any), as shown in Fig.9. The outputs from all eight 1-bit voters are then concatenated/merged to form a 16-bit input to a diagnostic unit. The diagnostic unit checks if different PEs fail for different bits. If yes, it signals the test to fail; otherwise, it signals the test to pass. In addition it returns the ID/code of the failing PE (if one exists). Finally, as shown in Fig.9, our word voter has an additional output, the actual majority output word – this additional output is required only if one wishes to implement self-repair, see Section 6.2.

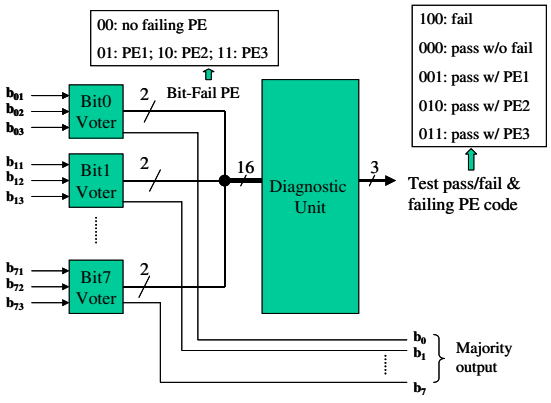

Fig. 9. 2-stage LUT-based 8-bit word voter.

The complexity of the word voter is as follows. Each 1-bit majority voter is implemented as an LUT with eight entries (note that we use three input bits, b_{i1}, b_{i2} and b_{i3}, as look-up address for the table associated with ith 1-bit voter), and each such entry stores three bits: two bits encode the failing PE for ith bit, and one bit gives the majority output. The diagnostic unit is implemented as one LUT with 2^{16} entries, where each entry stores three bits, one to encode pass/fail information and two to encode the failing PE. The total number of bits in the word voter is thus $3*8*8+3*2^{16}$. Note that this is about 1/3 of the complexity of the LUT for an 8-bit addition (which requires $9*2^{16}$ bits), which is one of the arithmetic/logic operations supported on a PE.

Although this overhead is relatively small, one may further reduce the total number of bits in the LUTs required by the word voter, by cascading diagnostic units, see Fig.10. In this design, the voting is still preformed in one stage (exactly as before, see Fig.9), yet the diagnostics are now implemented in three stages. The first diagnosis stage contains four 2-bit diagnostic units, each implemented as a 16-entry LUT (with three bits stored in each entry, as before). The second stage uses two 4-bit diagnostic units, each implemented as a 64-entry LUT with three bits entries (encoded as before), and the third stage uses a single 8-bit diagnostic unit, implemented as a 64-entry LUT, with again, 3-bits per entry. Thus, the total number of bits in the LUTs of this cascaded voter design is only $3*8*8+3*4*16+2*3*64+3*64=960$, i.e., it requires only 0.5% of the bits used by our previous (non-cascaded) design. When compared with the previous design, the cascaded 4-stage arbiter requires much less storage, yet uses more complex interconnects. Note that a similar cascading method can be used to implement a PEs' standard arithmetic/logic operations such that, again, less storage is required at the expense of more complex interconnects. Selecting the best implementation will depend on the defect characteristics of a specific nanotechnology.

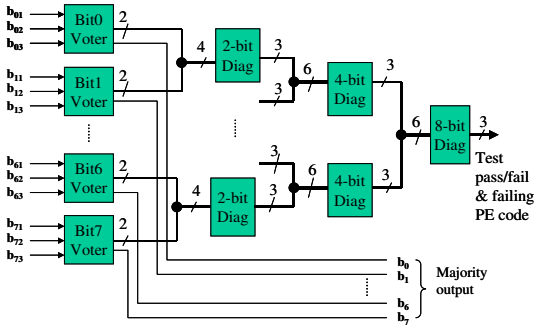

Fig. 10. 4-stage LUT-based word voter.

B. Step 2 – Testing Functional LUTs

As mentioned above, during the second test step, we scan the entries of all functional LUTs in the TMR tile's three peer PEs, and use the word voter of the arbiter PE to identify defective PEs (i.e., PEs with incorrect LUT entries). Fig.11 shows the testing circuitry used to scan all LUT entries. The 16-bit incrementer used by each peer PE is responsible for incrementing the index to the PE's LUT entries, and is designed such that it will increment only after receiving a pass from the voter – observe that, as shown in Fig.11, the "pass/fail" signal generated by the voter is fed back to the enable port of the incrementer. If for any entry (i.e., operation result), the voter detects an error (i.e., there is no majority), the scan test will fail.

Our implementation of the 16-bit incrementer uses two cascaded LUTs (not shown in Fig.11), where the first stage LUT implements an 8-bit adder with one input fixed as one, and the second LUT implements an 8-bit adder with the carry output from the first LUT being one of the inputs. Accordingly, its size is $9*2^9+8*2^9=17*2^9$ bits (i.e., 1.5% of a full 8-bit adder).

In our current implementation, if a PE is the minority for a particular LUT entry, and another PE is the minority for a different

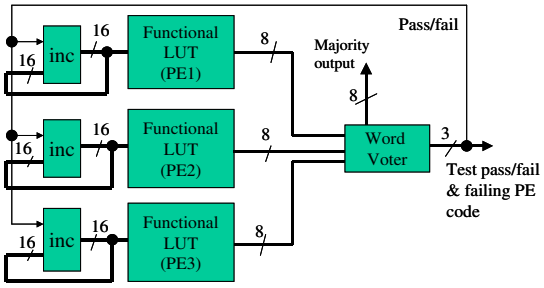

Fig. 11. Exhaustive testing of LUT entries.

LUT entry, the scan test will fail. We are thus favoring simplicity in the support circuitry – namely, this simple decision process can be implemented using a single register in the voter, which records the failing PE, and a two bit comparator. However, as an enhancement to the design, self-repair capabilities may be added. This can be done by incorporating extra self-repair control logic (possibly implemented using LUTs as well) in the feedback loop from the voter output to the enable port of the incrementer. The inputs to the self-repair control logic would be the "pass/fail" signal from the voter, and the encoding of the failing PE. Its outputs are the enable signal for the incrementers and the write enable signal for all the three LUTs. In this more complex implementation, once the voter detects an incorrect entry in a PE's LUT (i.e., the other two LUTs are identical/correct), it can enable writing back the majority output to the incorrect entry of the LUT. The incrementer will then be halted, by resetting the enable signal, and the entry would be rescanned after the repair occurs. If the repair is successful, the incrementer would resume incrementing, thus selecting the next entry. If the repair fails (indicating a hard fault, e.g., a 'broken' crossing switch), the voter stores the ID of the failing PE, as before. Although adding self-repair features is appealing, the impact on actual yield of incorporating this more complex circuitry on such simple PEs is not obvious, and is likely to be highly dependent of the specific fault regimes of the target technology.

VII. Experimental Results and Analysis

A. Experimental Methodology

We evaluated the proposed approach using the set of benchmark kernels shown in Table I. The flow cover adopted for each of the kernels is given in the table – e.g., the dataflow graph (DFG) of the two-dimensional discrete-cosine-transform (DCT) kernel is covered by four basic flows of type ft_6 (see Fig.3). Note that, for each kernel, there are many possible flow covers, yet for the experiments that follow, it suffices to select a 'good' cover. Moreover, for each of the kernels (and associated cover), we generated a component design with maximum potential configuration capacity, i.e., we mapped each basic flow to an individual mapping unit (MU). For instance, the component design for the DCT kernel consists of 4 MUs, each implementing a basic flow of type ft_6.

For each kernel, we first used extensive Monte Carlo simulation to estimate the probability of successful configuration on a region of each basic flow used on that kernel's cover (i.e., *flow yield*), assuming distinct defect regimes (P_e, P_a, P_c). Recall that P_e, P_a, P_c denote the probabilities of failure for processing elements (PEs) operating as a generic processor, PEs operating as arbiters, and connections, respectively. Specifically, given a tuple (P_e, P_a, P_c), we generated a large number of defect realizations on a region, using the TMR-based group testing algorithm described in Section V-A to obtain a (partial) defect map for each such region instance. We then used a simple table-look-up algorithm to find a feasible configuration for each basic flow used in the kernel's cover on that region. In this way, the yield for each such basic flow was estimated with an adequate confidence level.

TABLE I
Benchmark kernels.

Kernels	DFG Cover with basic flows
FIR Filter unrolled (FIRu)	$ft_6, ft_6, ft_6, ft_6, ft_6, ft_6, ft_6, ft_6$
Auto-Regression Filter (AR)	$ft_2, ft_4, ft_4, ft_2, ft_4, ft_4$
Avenhous Filter mod (AF2)	$ft_3, ft_3, ft_2, ft_{10}$
Avenhous Filter (AF1)	$ft_3, ft_4, ft_2, ft_{10}$
2D-DCT (DCT)	ft_6, ft_6, ft_6, ft_6
FIR Filter (FIR)	ft_6, ft_6, ft_6, ft_6
FFT	ft_5, ft_5

Yield at the next level of hierarchy, i.e., MU level, was then computed assuming an independent distribution of defects across regions. The basic component designs considered in all our experiments correspond to the special case of m *identical* basic flows being mapped into n regions in an MU with $m = 1$. The yield, i.e., probability of successfully configuring the m flows, is directly given by:

$$P_{MU} = \sum_{i=m}^{n} \binom{n}{i} P_r^i (1-P_r)^{n-i} \quad (1)$$

where P_r is the estimated yield for the basic flow on a region, obtained as discussed before.

Finally, yield at the component level was computed. Specifically, the probability of failing to configure all of the kernel's flows on a component containing k MUs, i.e., one minus yield, is given by

$$P_f = 1 - \prod_{i=1}^{k} P_{MU_i} \quad (2)$$

where P_{MU_i} is the yield of the ith mapping unit of the component.

B. Experimental Results

Our first experiment aims at determining the maximum achievable yield for each of the kernels, using our defect mapping technique. Fig.12 shows the probability of failure to configure each of the considered kernels on a corresponding component – P_f, when varying the number regions provided per MU, under defect regimes of $(P_e, P_a, P_c) = (20, 10, 2)\%$. As shown in the figure, all the kernels can be reliably configured on the target nanofabric with a very small failing probability, even when the defect density of PEs is as high as 20%, suggesting that our testing method provide sufficient coverage for the target nanofabric. Note also that the failing probability decreases exponentially with the number of regions in an MU, i.e., as capacity increases.

Clearly, the granularity/complexity of a PE will directly impact its failing probability, i.e. P_e, and in turn yield. (Recall that our PEs comprise a set of LUTs implementing standard 8-bit arithmetic/logic operations.) Fig.13 shows the probability of failure to configure each kernel when P_e varies from 1% to 20% (where $P_a = 0.5P_e$ and $P_c = 0.1P_e$), assuming that each MU contains the maximum number of regions (i.e., nine [11]). The figure clearly exhibits the significant impact of P_e on yield for these kernels, suggesting the critical importance of keeping the granularity and complexity of PEs as low as possible, as well as the potential need to incorporate redundancy into PE designs in order to place P_e in an acceptable range.

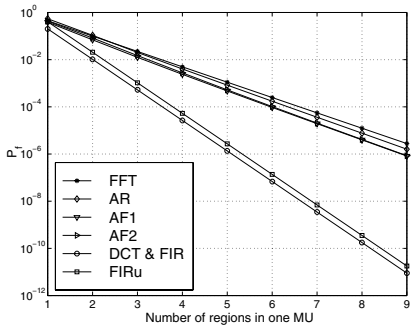

Fig. 12. Probability of failure to configure the kernels when $(P_e, P_a, P_c) = (20, 10, 2)\%$.

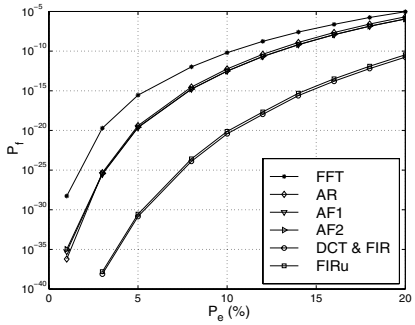

Fig. 13. Probability of failure to configure the kernels when varying P_e.

Finally, we analyze the impact of the complexity of the arbiter circuitry. Fig.14 shows the probability of failure to configure each kernel when varying P_a, i.e., probability of failure of a PE operating as an arbiter, from 1% to 20%, with fixed $P_e = 10\%$ and $P_c = 1\%$, assuming again that each MU contains the maximum number of regions. As shown in the figure, the yield shows little sensitivity to variations on P_a when P_a is less than P_e. Indeed, in our group testing method, each arbiter PE in a tile will be re-tested multiple times in other test tiles. Thus, false negatives resulting from faulty arbiters in a tile will most likely be corrected by other tiles. In this way, the 'arbiter reliability bottleneck' issue in traditional TMR (i.e. the reliability of a TMR output is bounded by that of the arbiter) is circumvented. However, as also shown in the figure, when P_a gets larger than P_e, it will eventually compromise the component yield. Therefore, it is still important to keep the arbiter design not overly complex, see Section VI. Moreover, the result suggests a minimal granularity for the PE circuitry, i.e., the complexity of the latter should be lower bounded by the complexity of the arbitration circuitry.

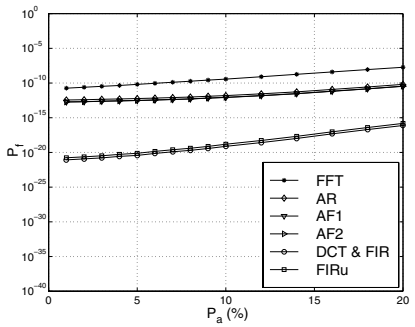

Fig. 14. Probability of failure to configure the kernels when varying P_a.

VIII. Conclusions

We have implemented a novel TMR-based group testing technique, aimed at enabling the design of defect tolerant nanosystems via reconfiguration, on appropriately architected memory-based nanofabrics. Possible designs for the required support circuitry have been presented and analyzed. Critical trade-offs between testing coverage and complexity have also been discussed. The effectiveness of the our defect testing approach has been experimentally demonstrated on representative benchmark kernels. Our future work includes developing the infrastructure for bootstrap programming of the PE's look-up tables and investigating the impact of adding self-repair capability on the nanofabrics.

References

[1] C. P. Collier *et al.*, "Electronically configurable molecular-based logic gates," *Science*, vol. 285, pp. 391–94, July 1999.

[2] T. Rueckes *et al.*, "Carbon nanotube based non-volatile random access memory for molecular computing," *Science*, vol. 289, pp. 94–97, 2000.

[3] Y. Cui and C. M. Lieber, "Functional nanoscale electronic devices assembled using silicon nanowire building blocks," *Science*, vol. 291, pp. 851–853, 2001.

[4] SEMATECH, "International Technology Roadmap for Semiconductors," 2004, http://www.itrs.net/Common/2004Update/2004Update.htm.

[5] S. K. Shukla and R. I. Bahar, *Nano, Quantum and Molecular Computing: Implications to High Level Design and Validation*. Kluwer Academic Publishers, Boston, MA, 2004.

[6] J. R. Heath *et al.*, "A defect-tolerant computer architecture: Opportunities for nanotechnology," *Science*, vol. 280, pp. 1716–21, June 1998.

[7] G. Bourianoff, "The future of nanocomputing," *Computer Magazine*, pp. 44–49, Aug. 2003.

[8] W. B. Culbertson *et al.*, "Defect tolerance on the Teramac custom computer," in *Proc. IEEE Symp. FPGA's for Custom Computing Machines*, 1997, pp. 116–123.

[9] S. C. Goldstein and M. Budiu, "Nanofabrics: Spatial computing using molecular electronics," in *Proc. Int. Symp. Computer Architecture*, Jul. 2001, pp. 178–191.

[10] A. DeHon, "Array-based architecture for FET-based nanoscale electronics," *IEEE Trans. Nanotechnology*, vol. 2, no. 1, pp. 23–32, 2003.

[11] M. F. Jacome, C. He, G. de Veciana, and S. Bijansky, "Defect tolerant probabilistic design paradigm for nanotechnologies," in *Proc. Design Automation Conf.*, 2004, pp. 596–601.

[12] C. He, M. F. Jacome, and G. de Veciana, "A reconfiguration-based defect-tolerant design paradigm for nanotechnologies," *IEEE Design & Test of Computers*, vol. 22, no. 4, pp. 316–326, July-August 2005.

[13] J. Han and P. Jonker, "A system architecture solution for unreliable nanoelectric devices," *IEEE Trans. Nanotechnology*, vol. 1, no. 4, pp. 201–208, 2002.

[14] ——, "A defect- and fault-tolerant architecture for nanocomputers," *Nanotechnology*, vol. 14, pp. 224–230, 2004.

[15] M. Mishra and S. C. Goldstein, "Defect tolerance at the end of the roadmap," in *Proc. Int. Test Conf.*, 2003.

[16] M. M. Ziegler *et al.*, "Scalability simulation for nanomemory systems integrated on the molecular scale," *Ann. New York Academy of Science*, vol. 1006, pp. 312–330, 2003.

[17] A. DeHon, P. Lincoln, and J. E. Savage, "Stochastic assembly of sublithographic nanoscale interfaces," *IEEE Trans. Nanotechnology*, vol. 2, no. 3, pp. 165–174, 2003.

[18] Z. Zhong *et al.*, "Nanowire crossbar arrays as address decoders for integrated nanosystems," *Science*, vol. 302, pp. 1377–1379, November 2003.

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

[20] D. Z. Du and F. K. Hwang, *Combinatorial group testing and its applications, 2nd edition*. World Scientific Publishing, Singapore, 2000.

[21] J. G. Brown and R. D. Blanton, "CAEN-BIST: Testing the nanofabric," in *Proc. Int. Test Conf.*, 2004, pp. 462–471.

[22] S. Mitra and E. J. McCluskey, "Word-voter: A new voter design for triple modular redundant systems," in *Proc. IEEE VSLI Test Symp.*, 2000, pp. 465–470.