To appear, IEEE International Workshop on Defect and Fault Tolerant Nanoscale Architectures (NANOARCH 2006)

Design and Evaluation of Fail-Stop Self-Assembled Nanoscale Processing Elements

Jaidev P. Patwardhan†, Chris Dwyer‡, and Alvin R. Lebeck†
{jaidev,alvy}@cs.duke.edu, dwyer@ee.duke.edu

†Department of Computer Science
Duke University
Durham, NC 27708

‡Department of Electrical and Computer Engineering
Duke University
Durham, NC 27708

Abstract

The semiconductor industry is exploring various device and manufacturing techniques to continue scaling transistor sizes beyond the capabilities of CMOS. This scaling is desirable, as it helps reduce device power consumption and area, while allowing higher operating speeds. However, the scaled transistors are increasingly susceptible to manufacturing defects. Architectures that are built using these transistors will need to tolerate defect rates that are orders of magnitude higher than those found in current CMOS technologies. We previously demonstrated an approach that provides defect isolation in a network of a large number of simple self-assembled computational nodes. This scheme can handle up to 30% defective nodes, but requires these limited size nodes to implement fail-stop behavior. In this paper, we explore trade-offs in implementing test mechanisms to achieve fail-stop behavior in nodes while meeting manufacturing constraints.

We use hardware self-test mechanisms to verify critical node components, and software tests for non-critical components. We reuse test logic where possible and move non-critical verification to software to meet technological size constraints. The modularity of the node and test logic, and the ability to disable defective components enables the use of nodes with some (non-critical) defective components. This allows the system to tolerate higher transistor defect rates. In particular, if nodes with at least one communication unit and one compute unit, or two communication units are allowed to operate, we can tolerate a transistor defect probability of $1.5x10^{-4}$. This is an order of magnitude higher than the defect probability that can be tolerated when a single defective transistor results in an unusable node.

1 Introduction

CMOS scaling is expected to reach its physical limits within the next two decades. This has led to an exploration of new technologies that could extend Moore's law beyond the capabilities of CMOS. DNA-guided self-assembly of nanoelectronic components is one such promising technology that could usher in an era of tera to peta-scale integration. A key advantage of this technology is its ability to manufacture a large number of circuit blocks in parallel. We previously proposed a circuit architecture [6] that could be used to manufacture computing circuits using this technology. However, these circuits are expected to be more susceptible to manufacturing defects and a system architecture that uses this technology must explicitly incorporate defect tolerance strategies. We previously presented a mechanism to isolate defective nodes in a random network of self-assembled nodes [7]. However, this assumes fail-stop nodes that are defective if there is a single transistor defect, which leads to unusable systems if the per device defect probability is greater than $4x10^{-5}$.

This paper explores strategies for implementing fail-stop processing elements (nodes) within the constraints imposed by self-assembly. We extend the defect isolation mechanism to operate within a node and use a combination of hardware and software test strategies to verify the operation of node components. If a node component fails or never completes the test, it is assumed to be defective and is not used, resulting in fail-stop behavior. Distinct tests for different node components enable the use of nodes with some defective components, as long as the defects do not affect critical functionality. Partially functional nodes can help the system tolerate a higher transistor defect probability ($1.5x10^{-4}$) and improve system connectivity as node defect rates increase. The primary contributions of this work are:

- implementing fail-stop nodes using a combination of hardware and software test strategies to verify the operation of node components, and
- extending a previously proposed defect isolation mechanism to improve system connectivity and increase tolerance of higher device defect probabilities.

The rest of this paper is organized as follows. We provide a brief description of our system and defect isolation algorithm in Section 2. We describe node architecture and test strategies in Section 3 and evaluate them in Section 4. We present related work in Section 5 and conclude in Section 6.

2 Defect Isolation using Reverse Path Forwarding

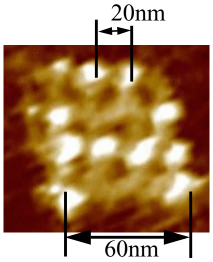

FIGURE 1. Patterned letter 'A' on DNA-lattice [5]

In this section, we provide a brief summary of a target system with our defect isolation mechanism. One of the key advantages of self-assembly is its ability to construct a large number of devices in parallel. We previously proposed a circuit and system architecture [6,9,8] that uses DNA-guided self-assembly to build a patterned scaffold (see Figure 1), with selective placement of carbon nanotube transistors as active devices. This enables the construction of a large number of small circuit blocks (nodes) that can be connected using metallized DNA to form a random network. Current limitations of self-assembly constrain node size, and we assume nodes with about 10,000 usable transistors. While the self-assembly process provides us with a great degree of control during design, it provides little to no control during the assembly process itself. This can lead to defects in the interconnect as well as within nodes. Figure 2 shows a small network of nodes with defective nodes and links. This network interfaces with the external world through a metal interface called a 'via'. Each via covers multiple nodes but is controlled through a single 'anchor' node. In the rest of the paper, we use the term 'anchor' to refer to a via/anchor pair.

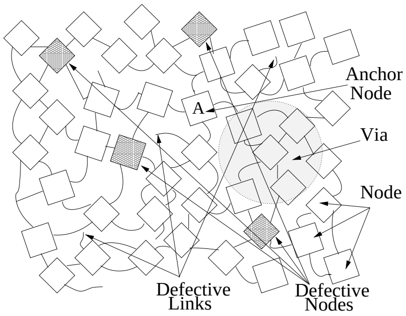

FIGURE 2. Random network of nodes with defective nodes and links (not to scale)

We adapt the Reverse Path Forwarding (RPF) algorithm [3] to isolate defective nodes in the network and organize functional nodes [7]. While a detailed discussion of the defect isolation is found elsewhere [6,7], we present a brief overview here. The algorithm begins with a single "gradient" packet inserted through an anchor. A node receiving this packet for the first time performs two actions: (a) it notes the input link on which the packet arrived (i.e., the gradient) and (b) it forwards the packet on all its active links, except the input link. If a node receives the packet again, it simply discards it. This results in a rapid broadcast of the packet to all nodes in the system. A node that receives the packet has a known route to the anchor where the packet was inserted by following the gradient through intermediate nodes. At the end of the algorithm, all functional nodes that received the gradient are connected on a tree (broadcast tree). We make a key assumption in this process - nodes that propagate the broadcast are defect free (or, defective nodes are fail-stop and don't participate in the broadcast). This results in the isolation of defective nodes (which don't propagate the broadcast) since no other functional node has a route to the anchor through a defective node. If defective nodes propagate the gradient broadcast, the system could be mis-configured and not function correctly. Thus, each node must implement fail-stop behavior.

3 Fail-Stop Nodes

In this section, we explore hardware and software test strategies that can help achieve fail-stop behavior. A node can be divided into three main components: a) communication logic, b) configuration logic and c) compute logic, and we develop independent test strategies for each. This simplifies test logic and enables the use of a partially functional node by isolating components that do not pass logic tests. In previous work, we made the conservative assumption that a node with a single defect is unusable. The ability to use partially functional nodes allows us to develop different node failure modes that can better utilize the defect-free parts of a node.

We begin the section with a description of node architecture (Section 3.1) and identify logic blocks that are critical to achieving fail-stop behavior (Section 3.2). We examine different hardware/software design options for implementing fail-stop, and identify the benefits of each approach (Section 3.3). Next, we describe the test mechanisms we use for communication (Section 3.4), configuration (Section 3.5) and compute logic (Section 3.6). We explore the effect of using such partially defective nodes on device reliability requirements, defect isolation, and system operation (Section 3.7).

3.1 Node Architecture

Each node is an asynchronous circuit and can be divided into three parts: communication logic, configuration logic and compute logic. The communication logic consists of four transceivers that allow the node to communicate with up to four neighbors on single bit links. Transceivers use a four-phase handshake protocol for data transfer over links. Each handshake transfers one bit, and links support full-duplex data transfer. Each transceiver supports three virtual channels [4] using 1-bit buffers. The four transceivers are connected to each other and the compute/configuration logic through point-to-point links for each virtual channel. The configuration logic is responsible for setting up internal node routing during the gradient broadcast phase. It config-

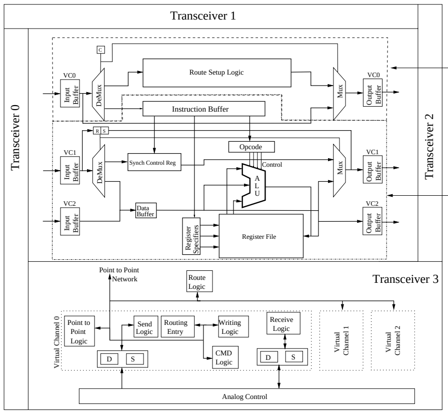

FIGURE 3. Block Diagram of Node (all transceivers are identical, but only transceiver 3 is shown for clarity)

ures virtual channel 0 (VC0) for broadcast routing, virtual channel 1 (VC1) for a depth first traversal of the gradient broadcast tree, and virtual channel 2 (VC2) with the reverse routing order of VC1. The computation logic implements a simple two-stage pipeline that allows the execution of instructions on 2-bit data slices. The computation logic has 32 bits of storage configured as a 16-entry 2-bit wide register file and an ALU that can perform simple arithmetic and logic functions. Figure 3 shows the block diagram of a node, clearly identifying the communication, configuration and compute logic (we show details of only one transceiver). A significant fraction of the node logic is devoted to communication support (both inter-node and intra-node). Next, we examine each logic block to determine its criticality with respect to achieving fail-stop behavior.

3.2 Critical Node Logic

We designate a logic block that must be defect free for the node to function correctly as being "critical". These logic blocks must be tested before a node accepts any external input to avoid the possibility of system misconfiguration. Logic for VC0 (communication logic) and route setup (configuration logic) is critical. All other logic in the node can be tested during the defect isolation phase since it does not affect the ability of a node to receive and send data. While this remaining logic is not critical, it must still be tested to ensure correctness. This can be performed with hardware or in software during defect isolation. Table 1 classifies various node logic blocks based on their criticality. The classification of logic blocks into critical/non-critical provides a simple way of determining what logic should be tested in hardware and what can be tested with software. Next, we explore different hardware and software test strategies.

3.3 Fail-Stop Node Design Options

Our goal is to achieve fail-stop behavior in nodes with minimal extra hardware. Critical logic must be tested before a node communicates with its neighbors, which implies the need for hardware test logic. For non-critical logic, we can choose between three options: a) hardware only test, b) hardware-software hybrid test, and c) software only test.

TABLE 1. Node Component Classification

Component	Critical	Description
Configuration Logic	Yes	Input arbitration on VC-0, depth first route setup
Transceiver Logic - VC0	Yes	Send/Receive logic for VC-0
Transceiver Logic - VC1/VC2	No	Send/Receive logic for VC-1
Point-to-Point Interconnect	VC0-Yes, VC1/2-No	Data interconnect within Node
ALU	No	Arithmetic Logic Unit
Register File	No	Register File in Compute Block
Instruction Buffer	No	First pipeline stage
Execution Control Registers	No	Storage for microinstructions

Hardware Test. We can add logic to each node to test the functionality of all components. This is equivalent to built-in self-test (BIST) [1,10] that does not require external test vectors. The primary advantages of hardware testing are low latency and the ability to test the node independent of the rest of the system. However, the size of a node that relies only on hardware test circuitry would exceed technological size constraints. This makes a hardware test strategy impractical. Note that critical logic still requires hardware testing.

Software (external) Test. For all non-critical logic, we could rely on software based testing using external test vectors. This can be combined with gradient broadcast to allow parallel testing of nodes, which would reduce test latency. This approach works well for instruction execution logic, but is not as useful for other components. For example, software testing of the transceiver circuitry for VC-1 requires hardware support to allow routing of test vectors to the transceiver logic. For small logic blocks, this extra hardware could be more expensive than implementing a hardware test scheme.

Hardware/Software Hybrid Test. The final option for testing, is to use a hybrid approach of hardware testing for simple components, and software testing for more complex components. For example, transceiver logic is simple, and requires identical testing for all three virtual channels. This can be done efficiently with simple test hardware. Furthermore, this test hardware can be shared between the three virtual channels. While this could increase test latency by a small amount, it results in reduced circuit size. Compute logic is fairly complex, and requires a large number of test vectors to ensure cor-

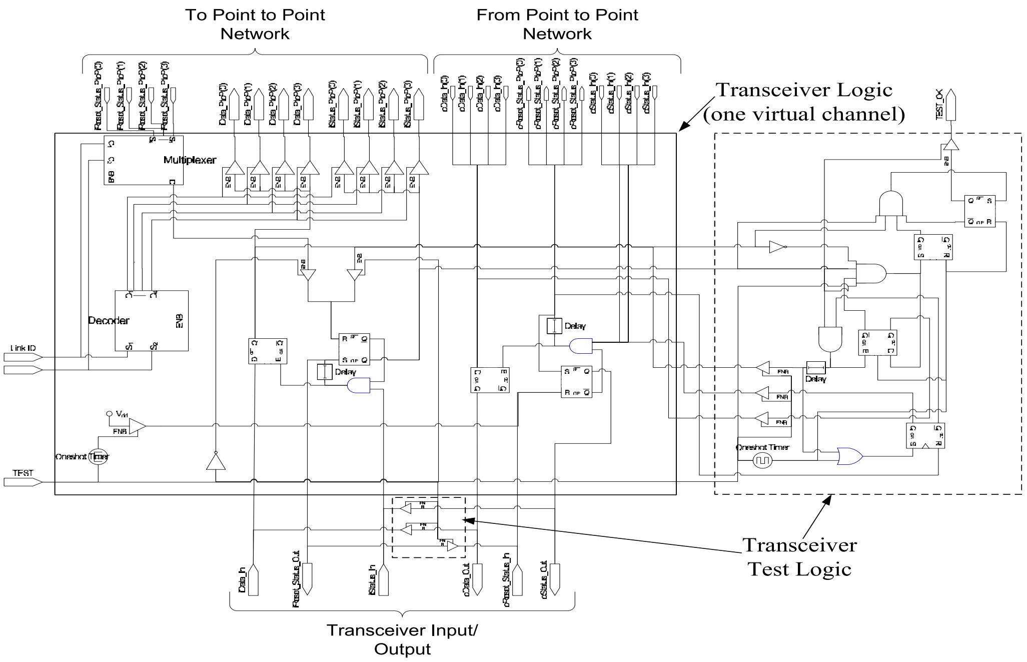

FIGURE 4. Transceiver logic for one virtual channel. (Test logic shown in the dotted rectangles is shared between virtual channels)

rect functionality. We can exploit existing hardware to test compute logic using external test vectors, with minimal extra hardware. This allows us to keep node size within technological constraints.

In summary, we use hardware test strategies for node components that can be tested with simple logic. If possible, we reuse test circuits to minimize overhead. In the next three subsections, we describe our test strategies for the three main components in a node.

3.4 Fail-Stop Communication Logic

Communication logic within a node supports three virtual channels and has two primary components: a) four transceivers, and b) point to point links. The circuits for VC0 are part of the node's critical logic since they are required during configuration. VC1 and VC2 are not part of critical logic, but can share test logic with VC0.

Each transceiver in a node must be tested to ensure correct functionality as defective logic in a transceiver can lead to incorrect system behavior. A node can be a useful part of a larger system even if it has only one functioning transceiver. However, if there are defective transceivers in a node, it is critical to isolate them from the rest of the system. To achieve this, we augment each transceiver with simple test logic and add a loopback path between the output and input logic of each transceiver. This path is enabled during test only. We exploit the simple four-phase handshake protocol used by the asynchronous logic in designing a test circuit that verifies the operation of the input/output logic. The transceiver is assumed to be defective by default. If the test verifies transceiver operation, the test circuit generates a signal to indicate that the transceiver is operational.

The largest component of the test logic is a two-bit state machine which inserts a test bit pattern into the transceiver output logic. The test pattern consists of two bits (0 followed by 1). The test logic inserts the 0, then waits until it loops back to the input logic. If the test logic successfully receives the 0 from the input logic, it inserts a 1 and waits for it to loop back. If both data bits (0 and 1) are received correctly, the test logic generates a "TEST_OK" signal. If the data is never received or incorrect data is received, this signal is not generated, isolating this transceiver from the rest of the node. Figure 4 shows the circuit for one virtual channel in a transceiver, along with test logic.

In addition to testing the transceiver logic, we need to test the point to point links that connect transceivers. However, routing on the point to point links depends on the result of the configuration process so we test point to point links when we test configuration logic.

3.5 Fail-Stop Configuration Logic

Configuration logic is responsible for determining the role of the node within the system, and for establishing communication routes (inter-node and intra-node). This makes the configuration logic an extremely critical component, and a node cannot function if it is defective. We use a hardware test mechanism that exploits transceiver logic to test the configuration block. Since it uses transceiver logic, the test occurs after the transceiver logic test. The test logic first configures the depth first traversal order of the transceivers within the node, skipping any transceivers that do not generate a “TEST_OK” signal. Next, the test logic uses a two-bit state machine to circulate a pair of bits (0 and 1) on all virtual channels. If the bits are routed correctly, they arrive back at the insertion point due to the loopback path at the transceivers. If the bits are received correctly, the node generates a “CONFIGURATION_OK” signal. To avoid masking defects due to defective route setup, each transceiver must ensure that each bit passes through it only once per VC. The configuration test fails if there is a routing error, the bits never return, or we receive the wrong bit values. A failed configuration test results in a node marked defective.

3.6 Fail-Stop Compute Logic

Verifying the compute logic in a node is not as critical as verifying the communication and configuration logic. This is because compute logic does not affect system configuration and a node with defective compute logic can be used to improve network connectivity. However, to ensure that the system generates correct results, the compute logic of each node must be tested. This test can be performed at any point before nodes are organized into larger computational entities. This allows us the flexibility of implementing hardware or software test strategies. In either case, the principle is similar to our previous test strategies - a successful test connects the logic block with the rest of the node. If the test fails, or does not complete, the block remains disconnected from other parts of the node.

Hardware Test. We can exploit existing logic to allow repeated execution of test instructions to verify the compute logic. However, this test is unlikely to cover all the logic in the compute block without significant extra hardware. Node size constraints and limited test coverage make this test strategy impractical.

Software Test. Software testing can be performed with minimal additions to the existing node logic. Testing of the compute logic must happen before nodes are organized into larger computational entities. We can combine software testing of the compute block with defect isolation by including the test vectors along with the configuration packet. Another advantage of software testing of the compute logic is the possibility of exhaustive testing to ensure correct operation.

Our choice of hardware testing for communication and configuration logic, and software testing for compute logic is driven by an analysis of the critical components of a node and technological constraints. As self-assembly technology matures, other test strategies could become more feasible. Next, we describe how we can exploit the modularity of the node to improve system connectivity and tolerate higher transistor defect rates.

TABLE 2. Node Failure Modes. C_xT_y defines the number of compute logic (x) and transceiver (y) failures that can be tolerated

Name	Description
C_0T_0	Node can tolerate no failures
C_0T_2	A node can tolerate up to two defective transceivers (compute logic must work)
C_0T_3	A node can tolerate up to three defective transceivers (compute logic must work)
C_1T_2	A node can tolerate defective compute logic as well as two defective transceivers
Hybrid	A node can tolerate C_0T_3 or C_1T_2

3.7 Using Partially Functional Nodes

In our previous work [7] we assumed that a node could either be defective or working correctly. However, if the probability of failure on an individual transistor is high, a larger number of nodes are rendered unusable. The test logic described earlier in this section opens up the possibility of using nodes with some defective components (if they do not affect system operation). For example, a node with a single defective transceiver can still communicate with up to three neighbors and perform computation. We explore four modes of failure that allow a node to operate with some defective components, defining each scheme based on the number of defects it can tolerate in the compute logic and transceivers. The failure modes are denoted C_xT_y, where x is the maximum number of defects that can be tolerated in compute logic (0 or 1), and y is the maximum number of defective transceivers that can be tolerated (0,1,2, or 3). The scheme used in our previous work cannot tolerate any defects and is denoted C_0T_0. The four modes we add are: C_0T_2 (a node cannot tolerate defective compute logic, but can tolerate up to two defective transceivers), C_0T_3, C_1T_2 and a hybrid of C_0T_3 or C_1T_2. We list these failure modes in Table 2. Each failure mode tries to include nodes that could contribute to system operation. The difference is in the minimum operating components each node must have to be used by the system. Nodes are considered useful under C_0T_3 as long as they have one functional transceiver and can be used to compute. Under C_1T_2 a node is useful as long as it has the potential to improve system connectivity by providing an extra path between two parts of the system (i.e., two active transceivers). The hybrid scheme includes nodes that can either perform computation, or provide an extra path between two parts of the system. As transistor failure probability increases, the number of nodes marked "defective" by each scheme increases. Simulations reveal that this increase is fastest for C_0T_0, and slowest for the hybrid failure mode.

Each node requires extra logic to operate with some defective components. This logic keeps track of defective components in the node and disables the node if the defects cross the failure threshold. For example, the C_1T_2 scheme requires six bits to keep track of the 6 primary node components (four transceivers, configuration logic, compute logic). In addition, it requires logic that determines if more than two transceivers have failed. While this adds to the size of the node, it allows us to better utilize each node.

3.8 Summary

We use a combination of hardware and software test methodologies to verify the operation of a node. We use hardware test logic for critical components, and rely on software testing for other components. A component can be used only if it undergoes a successful test. This results in fail-stop nodes as defective components are isolated from correctly functioning components. Since we use separate tests for node components, with a little extra logic, we can allow nodes to operate even if some (non-critical) components are defective. If transistor reliability is low, allowing these nodes to participate in the system should improve node connectivity. In the next section, we evaluate the effect of using partially defective nodes on the defect isolation mechanism.

4 Evaluation

We evaluate three aspects of our proposed scheme. First, we verify that the test logic for communication and configuration detects defects and measure the overhead of adding the test logic in terms of extra transistors required (Section 4.1). Next, we explore the relationship between device failure probability and the expected number of defective nodes in the system, in the context of different node failure modes (Section 4.2). Finally, we evaluate the benefit of our testing mechanisms by comparing how well the defect isolation mechanisms perform for different node failure modes (Section 4.3).

4.1 Test Logic

We implement the test logic described in Section 3.4 and Section 3.5 in VHDL and simulate it using the synopsys VHDL debugger. We first verify that the test circuit generates the "TEST_OK" signal in the absence of defects in the circuit within a deterministic delay. Next, we check the response of the test circuit when each signal within the circuit under test is forced to exhibit stuck-at behavior (i.e., forced to 0 or 1). In each case, we verify that in the presence of a stuck-at fault, the test logic does not return a "TEST_OK" signal. Since the test logic circulates a 0 and 1, we can detect stuck at faults on data lines. Since most data exchanges use handshake signalling, stuck at faults prevent the circuit from making forward progress (the handshakes require changes in the logic level). The test circuits increase the size of the communication and configuration logic by 18% (736 transistors) and 35% (248 transistors) respectively. The overhead for the configuration logic is higher since the original circuit is not very large.

4.2 Node Failure Modes

In this subsection, we explore the relationship between the transistor failure probability and defective nodes for different node failure modes (see Table 2). In our previous work [7], we showed that our defect isolation mechanism could tolerate up to 30% defective nodes. In that analysis, we assumed the C_0T_0 failure mode for a node, where 30% defective nodes correspond to a transistor failure probability of less than $4x10^{-5}$. It is unclear if self-assembly can guarantee such low transistor failure probabilities. We can tolerate a higher transistor failure probability by allowing nodes to operate with some defective components. We compute the expected number of defective nodes over a range of transistor failure probabilities, for different failure modes.

To study the relationship between per-transistor reliability and the fraction of defective nodes, we analyze a system with 10^6 nodes. Each node is assumed to have 10,000 transistors, with a uniform device failure probability (P_f). We use a uniform random number generator to generate random numbers (RND) in the interval [0,1]. Each random number corresponds to one transistor in a node. If RND<P_f, the transistor is defective. For each node, we compute whether it is defective for each failure mode. This analysis ignores defective interconnect (within and between nodes). For each value of P_f, we run 500 experiments with different random seeds.

In Figure 5, we plot the percentage of defective nodes in a system with 1 million nodes, as a function of the transistor failure probability. Each curve corresponds to one failure mode. As expected, the number of defective nodes in the system decreases as device reliability increases. However, we also see that the ability to test components within a node and allow graceful degradation allows us to reduce the number of defective nodes without increasing device reliability. It is important to note that for the hybrid failure mode, while a smaller number of nodes are designated defective compared to other failure modes, a large number of nodes have some defective components. While nodes with

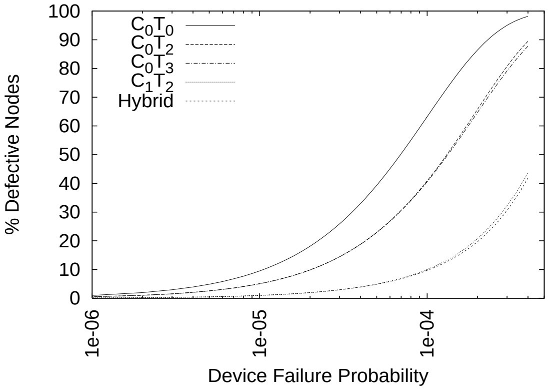

FIGURE 5. % defective nodes vs. device failure probability for different node failure modes

defective compute logic cannot be used to perform computation, they are useful in improving the connectivity of the network. Next, we use two baseline network topologies to evaluate the benefit of using partially defective nodes.

4.3 Defect Isolation with Partially Defective Nodes

In the previous subsection, we examined the effect of different node failure modes on the relationship between transistor failure probability and node defect rate. This analysis did not examine the effect of the location of defective nodes on the system. We now explore two different node topologies (random and grid) to determine the effectiveness of the defect isolation mechanism with partially defective nodes.

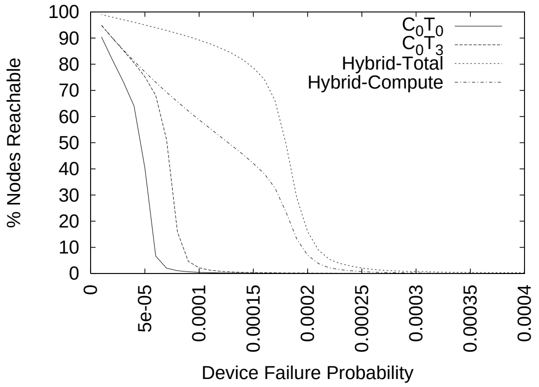

FIGURE 6. % Nodes Reachable vs. Device Failure Probability for a grid with different node failure modes

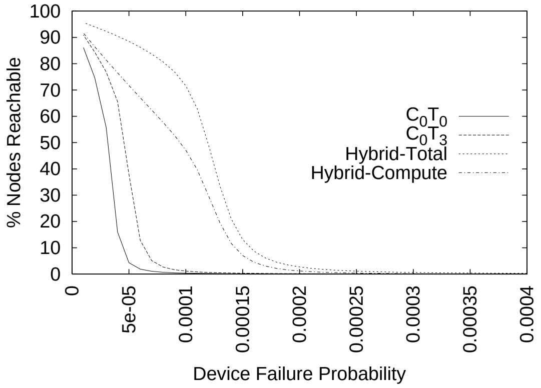

FIGURE 7. % Reachable Nodes vs. Device Failure Probability for a random network with different node failure modes

First, we compute the number of non-defective nodes that are reachable by the broadcast as a function of device failure probability, for three node failure modes (C_0T_0, C_0T_3 and Hybrid). We expect C_0T_0 to have the lowest number of reachable nodes, followed by C_0T_3, with the hybrid mode having the highest number of reachable nodes. However, a large number of nodes that are reachable with the hybrid mode, have defective compute logic and only act towards improving system connectivity. To account for this difference, we also plot the number of reachable nodes with operational compute logic. Figure 6 plots the average number of nodes (as a percentage of total nodes) that can be reached for the three failure modes as a function of device failure rate, when nodes are connected in a 100x100 grid. For each device failure rate, we use 100 seed values for the random number generate to generate different defect distributions, and compute the average of these 100 runs. From the figure, we see that the Hybrid failure mode delivers a significant advantage over C_0T_0 and C_0T_3 (even if we look at nodes with functioning compute logic only). While there is a sharp decrease in the number of reachable nodes beyond a certain device defect probability, this threshold is higher with Hybrid failure than with C_0T_0 failure.

We also compute the number of non-defective nodes that are reachable in a random network with 10,000 nodes. This random network is meant to be representative of self-assembled networks of nodes. The random network has inherently lower connectivity than a regular grid and some nodes might be disconnected from the rest of the network. This implies that we should see a reduction in the failure probability threshold for all schemes. For the random networks, we generate 100 random topologies, and then use 100 seed values per topology to get statistically accurate results. Figure 7 plots the average number of nodes (as a percentage of total nodes) that can be reached for the three failure modes. As expected, we see the knees in the curves have shifted left, but the general shapes are similar to those seen for a regular grid.

Finally, we evaluate the benefit of using the hybrid failure mode over C_0T_0. In Figure 8, we plot the average fraction of all nodes that are reachable as a function of the percentage of defective nodes as defined by C_0T_0 (i.e., single defect renders node unusable). We plot two

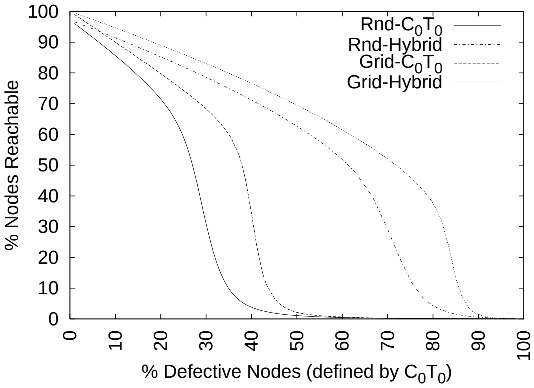

FIGURE 8. Effect of partial node

curves each for two types of network topologies (grid and random). The two curves correspond to the number of nodes reachable using C_0T_0, and the number of nodes with functioning compute logic reachable when using the hybrid failure mode. Note that the total number of nodes reachable by the hybrid mode is greater than those reachable with functioning compute logic, since the hybrid mode uses nodes with defective compute logic and two (or more) functioning transceivers. We see that the hybrid failure mode allows us to use nodes that would be unusable with C_0T_0.

4.4 Result Summary

Our results show that allowing partially defective nodes to participate in system operation increases the transistor failure probability that can be tolerated by the system. We have show that allowing nodes with defective compute logic, but functional communication logic to remain in the system improves network connectivity.

5 Related Work

There has been extensive work on testing circuits. The key difference in our work is the scale of the nodes, and the technological constraints that limit circuit size. Built-In-Self-Test (BIST) is a common technique used to test circuits. BIST strategies typically use linear feedback shift registers (LFSRs) to generate pseudorandom test patterns for circuits. While most BIST strategies are used with synchronous circuits, there are asynchronous BIST techniques as well [1,10]. Both the hardware test mechanisms presented in this paper are a form of BIST. We do not require (and cannot fit) large LFSRs since we deal with simple circuits and single-bit wide data paths.

The TERAMAC custom computer [2] from HP Labs is built out of a large set of partially defective FPGAs. It uses externally initiated testing to obtain an external defect map of the FPGAs. It then configures the system to isolate defective regions. Our proposed system is significantly larger than the Teramac, making defect map extraction infeasible.

6 Conclusions

In this paper, we present a scheme for achieving fail-stop behavior in limited size nodes by dividing them into modular components. We analyze the tradeoffs in implementing hardware/software test schemes for the components, and use hardware testing for critical node logic, and software testing for other logic. The use of partially functional nodes improves network connectivity, and helps the system tolerate devices with higher failure probabilities (increased from $4x10^{-5}$ to $1.5x10^{-4}$). As self-assembly matures as a technology, node size restrictions could reduce, allowing the use of faster, and more comprehensive hardware test schemes.

Acknowledgments

This work is supported by an NSF ITR grant CCR-0326157, the Duke University Provost's Common Fund, AFRL contract FA8750-05-2-0018, and equipment donations from IBM and Intel.

References

[1] V. C. Alves et al. A BIST scheme for asynchronous logic. In *Proc. of the Seventh Asian Test Symposium*, 27–32, Dec. 1999.

[2] W. B. Culbertson et al. The Teramac Custom Computer: Extending the Limits with Defect Tolerance. In *Proc. of the IEEE Intl. Symp. on Defect and Fault Tolerance in VLSI Systems*, Nov. 1996.

[3] Y. K. Dalal and R. M. Metcalfe. Reverse Path Forwarding of Broadcast Packets. *Comm. of the ACM*, 21(12):1040–1048, 1978.

[4] W. J. Dally. Virtual Channel Flow Control. *IEEE Trans. on Parallel and Distributed Systems*, 3(2):194–205, Mar. 1992.

[5] S. H. Park et al. Finite-size, Fully-Addressable DNA Tile Lattices Formed by Hierarchical Assembly Procedures. *Angewandte Chemie*, 45:735–739, Jan. 2006.

[6] J. P. Patwardhan et al. Circuit and System Architecture for DNA-Guided Self-Assembly of Nanoelectronics. In *Foundations of Nanoscience: Self-Assembled Architectures and Devices*, pages 344–358, Apr. 2004.

[7] J. P. Patwardhan et al. Evaluating the Connectivity of Self-Assembled Networks of Nano-scale Processing Elements. In *IEEE Intl. Workshop on Design and Test of Defect-Tolerant Nanoscale Architectures (NANOARCH '05)*, 2.1–2.8, May 2005.

[8] J. P. Patwardhan et al. A Defect Tolerant Self-Organizing Nanoscale SIMD Architecture. *(to appear) Intl. Symp. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Oct. 2006.

[9] J. P. Patwardhan et al. NANA: A Nano-scale Active Network Architecture. *ACM Journal on Emerging Technologies in Computing Systems*, 2(1):1–30, 2006.

[10] O. A. Petlin and S. B. Furber. Built-in self-testing of micropipelines. In *Proc. Third Intl. Symp. on Advanced Research in Asynchronous Circuits and Systems*, 22–29, Apr. 1997.