Defect-tolerant, fine-grained parallel testing of a Cell Matrix

Lisa J. K. Durbeck[#], Nicholas J. Macias[+]
Cell Matrix Corporation[*]

ABSTRACT

A fault testing methodology for a cell-based self configurable hardware platform (the Cell Matrix) is described. Background on the Cell Matrix is given, including its amenability to use despite the presence of manufacturing defects. The ability of cells within the Cell Matrix to isolate faulty regions is also described. A method for testing individual cells, based on an external test driver, is discussed. The benefits of locating this test driver inside the device under test are explained. A method is described for efficient, autonomous, robust creation of a network of self-testing structures (called Supercells) for parallel implementation and execution of this test driver. Sample tests are described, and their results are given, demonstrating the effectiveness and robustness of the testing methodology. A discussion of the research, including conclusions, is presented. Plans for future work are discussed.
Keywords: Fault testing, reconfigurable hardware, self configurable, process driver, nanotechnology, autonomous, fault tolerance, Cell Matrix

1. INTRODUCTION

The age of atomic-scale manufacturing is quickly approaching. In this coming era, machines will be assembled atom-by-atom, leading to the creation of structures many orders of magnitude smaller than those possible with today's technology[1]. These structures will include not only pure mechanical systems, but computing engines as well. One immediate application of such manufacturing techniques will be the creation of physically smaller versions of today's electronic devices (memories, CPUs, etc.), with associated savings in power, increases in speed, and so on. A less frequently discussed application will be the creation of physically large computing devices, containing many orders of magnitude more components than today's devices.

Such so-called Avogadro machines (those containing on the order of 10^{23} components) will likely require very specialized structures. For example, there is no reasonable way known today to expand a Pentium® processor to one with 10^{23} transistors. On the other hand, highly scalable structures, such as memories, are ideal candidates for such large-scale circuits.

For more general-purpose circuit building, reconfigurable hardware is an ideal enabling technology, allowing the implementation of non-scalable, custom ASIC designs on top of a fine-grained, atomic-scale reconfigurable substrate. However, most current reconfigurable architectures are not scalable. One exception to this is the reconfigurable architecture described in this paper: the Cell Matrix™. In contrast to most reconfigurable devices, the Cell Matrix has an infinitely scalable architecture. This scalability makes the Cell Matrix an ideal target for atomic-scale fabrication, since the architecture can readily make use of huge switch counts.

One important issue, which can not be neglected when discussing Avogadro machines, is the question of manufacturing defect rates. Even in today's silicon processes, yield rates have become a significant part of a device's final cost[2]. This is due in large part to the GO/NO-GO nature of today's test strategies. A series of tests are applied, and if a device fails a test, it is discarded. Xilinx's recently announced EasyPath™[3] improves the situation a little, by testing a subset of a device based on the intended application. However, it appears that this still results in a GO/NO-GO decision, at least for the given design.

In tomorrow's atomic-scale processes, this problem becomes much more severe. For example, a manufacturing defect rate of one fault per million million million units (1 in 10^{18}) would normally be considered very good. But, when applied to a device containing 10^{23} switches, one would expect 100,000 faulty switches inside the device. While eventually it may be feasible to manufacture perfect, defect-free devices, a more practical solution is to develop defect-tolerant devices, i.e., systems which can function properly despite the presence of manufacturing defects. The testing result thus

ld@cellmatrix.com; + nmacias@cellmatrix.com; * www.cellmatrix.com; phone (877) 473-0882

changes from GO/NO-GO to a grade, indicating "how defective" the device is. Here again, the Cell Matrix is particularly well suited to this paradigm, for a number of reasons:

- The Cell Matrix is a fine-grained reconfigurable device, in which larger components and subsystems are built from simple, elementary cells.
- The Cell Matrix does not contain critical, irreplaceable components. Rather, if a region of the device is damaged, identical hardware from elsewhere in the Cell Matrix can be used in its place.
- Cell Matrix hardware is inherently fault tolerant, due to the lack of global buses, and the presence of only nearest-neighbor communication among cells.
- The Cell Matrix is not only reconfigurable, but is self configurable, meaning that it can analyze and modify its own circuitry, autonomously, thus leading to powerful mechanisms for detecting and handling defects within itself.

It is this last property in particular that the current work exploits. Specifically, we address the question: "Given an Avogadro-scale Cell Matrix, in which one expects a non-zero number of manufacturing defects, how can we detect, isolate, and work-around those defects?"

A simple approach would be to use something similar to the scandisk program, which tests hard drives for defects. However, testing an Avogadro-scale digital circuit differs from this in two fundamental ways:

1. the system being tested is much larger. 10^{23} cells is 13 orders of magnitude (ten thousand billion) more than the number of bytes in a 10 GByte hard drive; and
2. for maximum usefulness, the system should be self-testing, i.e., we should not require a large, defect-free external controller which can orchestrate the testing of the Cell Matrix, since the Cell Matrix is supposed to be **the** device manufactured via atomic-scale fabrication. In other words, whereas scandisk runs on a perfect CPU and tests an external device, our approach must run on the Cell Matrix, **even as the Cell Matrix is itself being scrutinized for defects**.

Our approach to these two requirements was to employ parallel, self-testing techniques, taking advantage of the Cell Matrix's distributed, internal configuration capabilities. By utilizing parallel fault testing we can test on the order of n^2 cells in only n steps, assuming a two-dimensional Cell Matrix. For a three-dimensional matrix, this increases to order n^3 cells in n steps. Thus, testing 10^{23} cells in a three-dimensional matrix would require fewer than 50 million steps.

Another benefit of this approach to fault testing is the extreme robustness of the fault testing itself. Except for some edge cases, cell failures do not in general impair the ability of the system to test itself. Thus, a device containing many defective regions, in more or less any pattern, can still be efficiently analyzed.

Because the Cell Matrix exhibits fine-grained control over its own configurable components, it is possible to configure cells to form guard walls[4] isolating faulty regions and preventing them from impacting non-faulty regions. Moreover, once faulty regions have been so marked, it is possible for the Cell Matrix to autonomously configure itself into a final circuit, taking into account the (dynamic) location of faulty regions, and working around them as needed. Recent work on autonomous self-assembling circuits[4] utilizes this approach.

Finally, because configuration of a Cell Matrix occurs at run-time (there is no distinction between run-time and compile-time of a Cell Matrix), it is possible to develop circuits and strategies for handling not only manufacturing defects, but run-time faults as well.

2. BACKGROUND

This work is based on the Cell Matrix self configurable architecture[5,6]. A Cell Matrix is composed of a regularly connected collection of simple processors called cells. Each cell possesses a fixed number of inputs and outputs, which are connected to the outputs and inputs of its immediate neighbors, according to a fixed, system-wide topology.

Within each cell is a small memory, or lookup table, which describes how a cell's outputs should be set in response to every possible combination of input values. In a cell's normal mode of operation (called D-mode), a cell continually examines its inputs, and produces its corresponding outputs. A collection of D-mode cells can thus be made to operate as any desired digital circuit, combinational or sequential, by simple having each cell's lookup table appropriately configured.

There is another mode in which cells operate, called C-mode. When a cell is in C-mode, its inputs are used to supply new values for its lookup table, while its outputs transmit the previous contents of its lookup table. This is thus the mode in which a cell's lookup table can be read and written, i.e., the cell can be reconfigured.

Note that a cell's mode is a cell-level property, not a system-level one. In other words, each cell's mode is independent of the mode of other cells within the matrix. One cell may be in D-mode, while a neighboring cell is in C-mode. More generally, one set of cells may be operating in C-mode, being configured simultaneously, while another set of cells operate in D-mode, perhaps controlling the configuration of the first set.

The determination of a cell's mode is based on the outputs of its neighbors. Specifically, assuming each cell has four neighbors, then each cell has four D outputs and four C outputs. The D outputs and inputs are used to exchange data, which is acted upon as described above. The C outputs and inputs are used to affect a cell's mode: if any of a cell's C inputs are 1, then that cell is in C-mode; otherwise, the cell is in D-mode. Since a cell's C inputs are determined by neighboring cells' C outputs, it follows that any cell can cause a neighboring cell to enter C-mode, simply by asserting its own C output to that neighboring cell. For example, in Figure 1, cell x is asserting its Eastern C output, which is also cell y's Western C input. Thus, cell y is in C-mode, and therefore is sending its current truth table through its own Western D output, which can be read by cell x. Cell y is also reading its Western D input (which is written by cell x), and re-loading its own truth table from the input. Thus, cell x is reading and modifying the configuration of cell y.

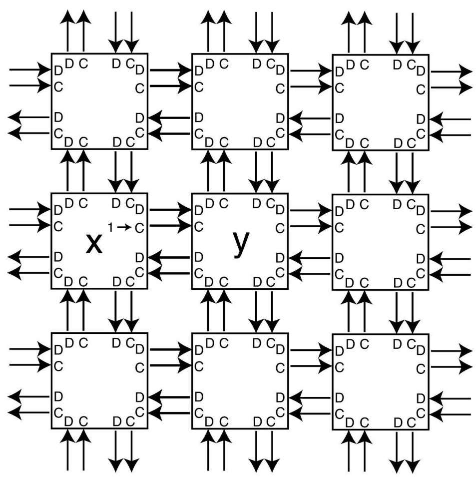

Figure 1. 3x3 Cell Matrix

This interaction of C and D outputs and inputs thus allows any cell to reconfigure any of its neighboring cells. Note, however, than a cell has direct connections only to immediately adjacent cells, and thus has no way to directly configure a non-adjacent cell. For example, in Figure 2a, cell x can modify cell y, but has no direct access to or control over cell z.

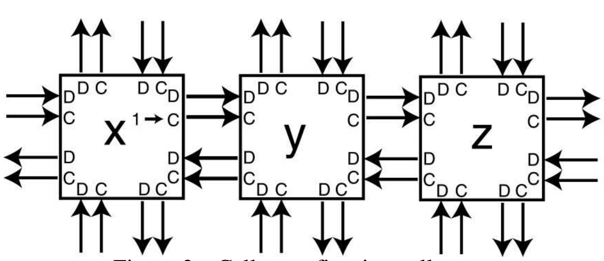

Figure 2a. Cell x configuring cell y

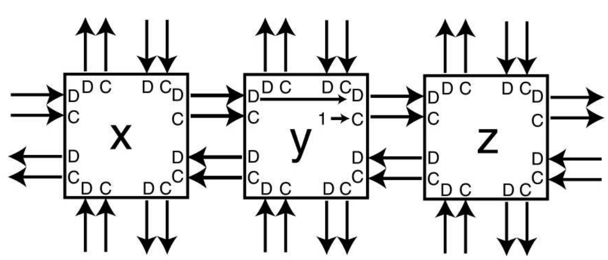

Figure 2b. Cell x configuring cell z

While cells can only **directly** affect neighboring cells' inputs, it is quite feasible for a cell to reconfigure a non-adjacent cell, by first configuring intervening cells, and then using those to configure the target cell. In Figure 2a, cell x is configuring cell y so that, as shown in Figure 2b, cell y will subsequently allow cell x to configure cell z. Specifically, cell y has been configured to assert its Eastern C output, thus placing cell z into C-mode. Cell y has also been configured to pass data from from cell x to cell z. In this configuration, cell x is thus able to load a new configuration into cell z, by having first configured cell y to cooperate with this remote configuration operation.

More generally, it is possible to develop structures called wires[7] which can be grown in linear time, and allow a set of cells to control non-adjacent cells throughout the matrix.

3. METHODS

Cell Matrix hardware can still be used even if it contains failed or faulty cells; however, before this is possible in practice, techniques must be developed to identify faulty cells and isolate them from the rest of the system, and configuration systems must be enhanced, either with techniques for using maps of faulty regions when placing and routing circuits, or with algorithms for using Cell Matrices with regions of hardware that have been removed from play by the test driver. One complete system with these characteristics is described elsewhere[4]. The present paper describes in greater detail the aspects of that work that led to fault diagnosis and fault isolation.

A test driver was designed and implemented for Cell Matrix hardware to enable the identification and isolation of faulty cells. Because a Cell Matrix is comprised solely of cells connected to their nearest neighbors, it is sufficient to design tests for individual cells and their interconnections and then perform those tests on all cells. The approach taken is to create a two-tiered design. The lower tier of the design is occupied by a mechanism to perform tests on cells that will turn up problems with cell function and interconnect function[#]. On the upper tier is a mechanism for addressing each cell of the hardware as fully as possible, despite the existence of faulty cells in the hardware.

The design goals for the test driver are thus as follows:

- It should permit reporting of faults with high resolution, in as limited a region around the fault as desired for the larger intended application of this particular piece of hardware. The region size will necessarily vary with the type of fault, and the fault's extent is recognized to be outside the control of this system to modify. However, conservatism in marking cells as faulty should be kept to a minimum. The fewer cells incorrectly marked faulty, the more usable hardware will be available. Also, the more precise the reporting of fault locations and types, the more likely the success in developing specific techniques to improve the manufacturing process.
- It should permit access to a region despite failed regions near it. It should also enable the prevention of faulty cells from either interfering with tests of other cells or cutting short the testing.
- It should be easy to extend to a decentralized, parallel, distributed fault testing process with as small a footprint as possible. The testing can later set up so that many testing apparatus can be deployed in parallel on separate regions of the hardware, enabling O(sqrt(n)) rather than the more typical O(n) testing times, so that very large Cell Matrices can be efficiently tested for faults, or more extensive tests can be run on each cell. Flexible methods of arriving at each cell are preferable so that faulty cells do not prevent the testing of cells further away. The smaller the testing apparatus, the more likely it can be deployed in a region containing faulty cells. It is undesirable to introduce a critical component during testing, and so the testing apparatus needs to be either highly flexible or constructed out of cells that have already been determined to be non-faulty.
- Ideally this design should lead to a test driver that can share the hardware with other tasks so that it can perform its functions on subcomponents of a critical system while that system is running.

The following is an incremental explanation of the methods used to test Cell Matrix hardware for faults.

3.1 Testing a single cell

First a definition is necessary of what is considered a failed or faulty cell. A failed cell does almost nothing that it should, as would be the case if it did not receive power; a faulty cell does some of the things it should and does not do other things it should, such as not giving the correct outputs for certain truth table configurations. It is beneficial but

#An analysis of the types of tests that should be used is not a focus of this paper; rather, the focus is on making it possible to conduct those tests once they are identified.

insufficient to simply detect failed or faulty cells: we must also be able to control the impact of these cells, because the larger goal is the ability to create perfect circuits and systems on top of imperfect hardware. Another goal is the gradual improvement of whatever hardware manufacturing techniques are used, such that fewer cells fail, and/or certain kinds of failure conditions can be eradicated.

Determining whether a cell does the things it should can be achieved by testing cell function. A cell can be represented by a circuit, and the function of that circuit can be tested using black box methods. In other words, the internals of the cell are not directly plied, and instead the cell's functional behavior is examined: given a particular input and state, the cell should produce a particular output. The test driver can then give the cell a set of inputs and compare the set of outputs against expected outputs. Even though this is black box testing, it can still provide some information about the internal workings of the cell when analysis of a pattern of failures reported during testing is combined with knowledge of the physical layout of the cell. For instance, given a particular physical layout of cell memory, a bitstream can be constructed that stores a 1 next to a 0 and then reads them back to capture unwanted interactions between two memory bit locations.

The object of the tests is to determine that the cell can be configured at all, and that it can be configured from all sides, and that the cell can be configured to implement any truth table, and that it can correctly perform any truth table and output results to all sides. A cell can be said to be operating perfectly if its functions are exhaustively tested, and it passes all the tests. The bulk of the testing time in exhaustive testing of a Cell Matrix cell is spent changing the cell's lookup table and making sure there are no odd interdependencies between memory locations leading to function loss. In practice, however, perfect operation will rarely be unequivocally established, because exhaustive testing is too time-consuming, particularly for runtime fault detection. Instead, testing can be limited to a very small set which generally should include:

- whether the cell can be configured from all sides
- whether the cell can communicate with all neighbors
- a small subset of all possible cell configurations, such as those most typically used. If the intended configurations of this cell when used in the field are known, testing can be limited to that small set.
- a set of tests that catches the kinds of hardware manufacturing errors likely as a result of using a particular manufacturing technique

Using the definition of a Cell Matrix cell, the four basic operations of a cell can be used to construct general categories for cell failure modes that can be useful for designing tests:

- a cell must receive power; if the hardware uses clocking (i.e., if the cell is in C-mode), then a clock signal must also be received and properly timed and shaped.
- a cell must be able to receive inputs from its neighbors and send outputs to its neighbors without any data loss or garbling, and thus the wires between cells must be functioning properly;
- C-mode operation must be tested to ensure that a cell is able to successfully program each of its neighbors and vice-versa: each neighbor must be able to program this cell;
- D-mode operation must be tested to ensure that a cell properly implements its truth table when in D-mode.

The set of tests should be developed so that it covers these four operational categories and finds failures in these four areas. For the present work, it was not necessary to distinguish among the four, only to make sure that we were able to cover all four categories. Additionally, it was not necessary to use a wide battery of tests, because our object was to develop the test driver, not the tests, nor was it to analyze hardware. Instead, we controlled the types of failures that we were experimenting with and could cause specific kinds of failures on otherwise perfect hardware, and then constructed specific tests for those kinds of failures.

Function testing that covers these operational categories for a Cell Matrix cell will typically involve putting the cell into a known state, such as one resulting from programming or configuring the cell's lookup table, then sending the cell a particular input stream and checking its output against an expected output stream. For the purpose of providing a simple exposition of the method used to test one cell, let us assume that the cell can be physically removed from the hardware and placed on a testing apparatus. The apparatus must provide access to all the cell's inputs and outputs as well as support circuitry needed for conducting the tests, such as a small comparator circuit that permits cell outputs to be compared to expected outputs. An input stream consisting of a sequence of triplets of information is constructed for testing the various functions of the cell. These triplets consist of: 1) a sequence of commands to program the cell that, if programming does not encounter any faults, will put the cell into a known state, followed by 2) input data that will be sent to the cell once it has been programmed, and 3) the expected output from the cell corresponding to each piece of input data. The expected output is sent to the comparator circuit and compared with the signal from the output wire of the cell. The series of programming commands and the input are sent to each of the input lines corresponding to each of the sides of the n-sided cell. This work was done with four-sided cells but was designed to be generalizable to cells with more or fewer connections. The test triplet is fetched and executed once for each input line of the cell, and during testing of a particular side, the appropriate C input is used in conjunction with its corresponding D input to configure the cell. Output from the side's corresponding D output is fed into the comparator circuit and compared bit by bit with the expected output. If the comparison fails, the output of the comparator circuit is high; otherwise, it is low. Information about which test failed is available as well, but for the current work this was not important and was not captured. Instead, the comparator circuit was designed so that once the failure flag was set, it remained high throughout the remainder of the cell test and was sampled at the end of the testing series.

A program was written that permitted easy definition of the desired triplets corresponding to a test for a certain cell function or a likely source of faults. The output of this program is a Cell Matrix configuration string that has the triplets embedded in it. Information embedded into this configuration string also controls the timing of the comparison of the cell's output to the expected output.

3.2 Multicell test sequence

It is necessary to describe how the testing apparatus is implemented. We cannot actually remove the cell from the matrix in order to test it, and typically the I/O wires of most cells in a matrix are not accessible from the outside, and so we cannot even directly access many cells for testing. However, it is possible to indirectly access and control every cell in a matrix from one of the accessible cells, such as a cell on the edge of the matrix that is hooked up to the I/O pins of the chip. Therefore from a point outside the matrix, it is possible to access all cells within the matrix by configuring cells to implement portions of a wire[7]. The wire is built through a series of configuration strings called a wire building sequence, all sent to the originating point where the wire begins. During the building process the wire can be made to head in any direction and turn in any direction.

Given this ability to create wires to allow communication between a fixed location and any remote location, one approach for implementing the testing apparatus is fairly straightforward to conceive of and will be described first, with the actual implementation — which is similar in style but has some critical refinements — described later. The testing apparatus can be located in circuitry external to the Cell Matrix hardware and hooked up to a small number of adjacent edge cells. Wire building commands can then be used from the interface point to create wires from the testing apparatus to any cell within the matrix. Because wires are made from cells, multiple pathways can be used to reach any given cell by creating wires that use different sets of contiguous cells. In this way, once a faulty cell has been detected, the wire can be rerouted around it and testing can continue. We did not implement rerouting at this level of the design, because the larger context of this work required a nonvariable configuration string, as described separately[4]. Instead, rerouting to avoid faulty cells was implemented as part of the upper tier of the design, which is described below.

Figure 3 illustrates how remote cell access is achieved. I/O wires are indicated by the repeated string of arrows extending to two different locations. Wires implemented from Cell Matrix cells are extended from the testing apparatus to two different cells within the hardware, the ones labelled with a '*' and colored dark grey.

A cell is tested from each side, and any failures are reported for the side and the first test that failed. The way that each side of a cell is reached has not yet been described. There are various ways to do this, and a way that minimizes the number of steps taken to test the matrix is preferred. It is possible to arrange for a cell-centric testing scheme that resembles the initial description of a cell removed from the matrix and placed within a testing apparatus, with the apparatus building wires to access each side of the cell and test it, then moving on to another cell. This could be achieved in a reasonably efficient number of steps without the construction of separate wires to each side if a wire were routed to one side of the cell and then around the perimeter of the cell, testing each side as it is reached. The wire could then be unfurled or cut and then the end rebuilt to advance to the next cell in the row or column. However, the unknown state of other cells in the matrix makes this approach difficult. The original design goals require that the test driver provide the most precise pinpointing of fault location possible. Therefore this method would require that the perimeter cells used to access this cell had themselves already been tested. It is hard to conceive of a testing pattern that would test the cells both above and below each and every particular row of cells, because that quickly leads to a paradox.

A side-centric and wire-location-centric testing scheme is more reliable and efficient. The stencil of this scheme is illustrated by Figure 4. The figure on the left shows which cell sides are tested, depicted as bold black edges. The figure on the right shows the set of cells used in the test, depicted as the set of dark grey cells. The region is controlled from a single point, the cell with the '*' symbol.

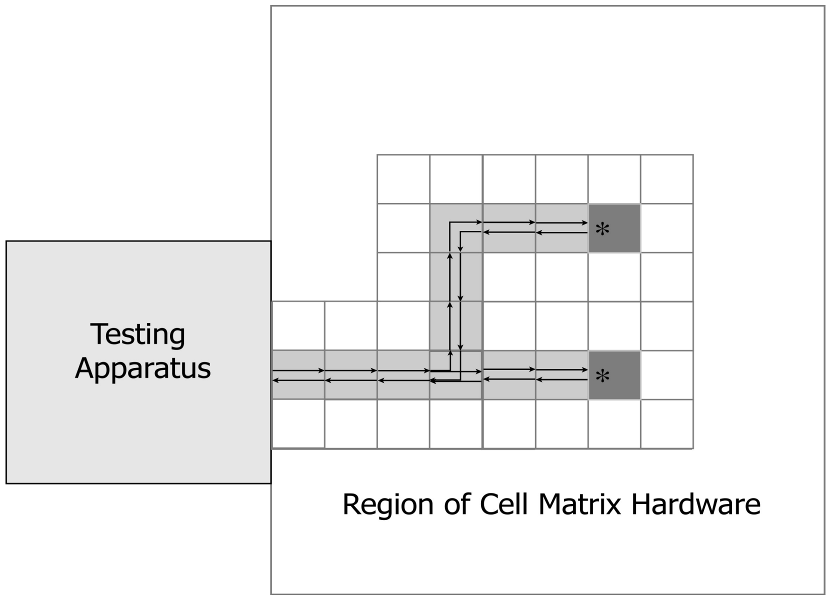

Figure 3. Remote Cell Access

This cell is accessed remotely via the I/O wire indicated by the arrows. This scheme tests all sides easily reachable from the end of the wire and then advances the wire to incorporate the tested cell. All sides of a cell are tested before the cell becomes part of the testing system, but they are not all tested at the same stage of the process or rung of the wire. The sides are reached from the main wire by the temporary construction of wire extensions that permit the desired cells to be treated as the head of the wire. Some of these involve bending the wire to reach cells above and below the main test cell. The sides tested can be chosen based on wire orientation such that even with a relatively simple variation on a raster scan type pattern, no cells are used as portions of wires until after they are fully tested from all sides. Thus the successive tests of cells in the hardware can be made to sweep from the starting point in a simple pattern. Another program was written to embed the test triplets configuration string into a longer configuration string that repeated the test pattern, but included wire-building commands that programmatically moved around the matrix, accessing cells and testing them. This program was then optimized for speed by the definition of efficient traversals of the hardware, so that the number of wire-building commands sent could be minimized, and also optimized by the removal of any repetitive tests of a cell as part of separate traversals of the hardware.

3.3 Upper Tier design and implementation

Combining all of the chosen implementations above provides an initial design and implementation for the upper tier test driver. The external test apparatus could be set up alongside Cell Matrix hardware and directly connected to a small number of adjacent cells in the matrix whose locations are known a priori, and a configuration string containing the test triplets can be generated and embedded repeatedly into a configuration string that builds wires to access each cell in the hardware according to a raster scan pattern and performs the tests on the sides illustrated in Figure 4.

This upper tier design has several shortcomings, all of which we addressed in our final implementation. First, if a faulty cell is encountered, depending on whether the fault affects its capacity as a piece of wire, it prevents the independent testing of all cells that use it as part of a wire. Two solutions to this are 1) to generate the configuration string dynamically so that when a faulty cell is encountered, future wire-building does not use this cell, or 2) to limit the size of the region for which this cell can prevent testing by breaking the hardware into subregions, each of which is accessed and tested independently. The second approach was taken because the project required a nonvariable configuration string. Also, breaking the hardware into separate subregions that could be independently tested permitted a solution to the second problem with this technique, namely that its performance is linear with the number of cells in the hardware and thus is slow for large Cell Matrices. The consequence for Avogadro computers is that although you might be able to construct them, you would never finish testing them. Independent subregions, on the other hand, can be tested in parallel. A design that broke the hardware down into fairly small regions and accessed many new regions at each time step would be far preferable for fast testing times. Third, implementing the testing apparatus as a separate piece of hardware is inefficient: the external testing apparatus is not guaranteed to be defect-free and would thus require a separate testing strategy and implementation. This was solved by putting the testing apparatus onboard the Cell Matrix hardware, on top of cells determined to be defect-free. This also permitted an inexpensive and simple way to achieve as many testing apparatus as needed for parallel tests of independent subregions of the hardware. The fourth shortcoming is similar to the first but more general than just wire-building: this approach does not isolate failed regions so as to prevent them from affecting the tests of other regions. This shortcoming was solved by implementing mechanisms that automatically walled off a region when a cell in that region was determined to be faulty.

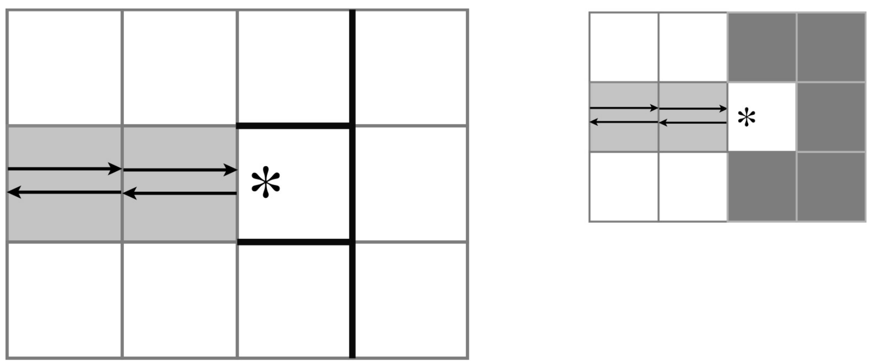

Figure 4. The set of cells involved at each leg of the cell test.

The improved implementation of the upper tier breaks the Cell Matrix into regions (called Supercells) of 44x44 cells, tests these regions, then lays down useful circuitry in that region (such as the testing apparatus) to permit testing of the untested regions beyond it as well as coordination among Supercells. The Supercell design tests subregions in parallel, with multiple test apparatus running simultaneously, enlarges the number of 44x44-cell regions under test (RUT) at each step, provides multiple paths to each 44x44 subregion via an ever-growing Supercell network, and isolates regions that contain one or more faulty cells so that they cannot prevent the Supercell network from growing to cover the full extent of connected, nonfaulty regions on the hardware.

An initial step in the construction of a Supercell is the systematic testing of each cell in the RUT for faults. This involves getting to each cell within the region and testing it for faults from all sides. A program was written to generate successive wire-building commands to reach all cells and invoke the configuration string for the test triplets for all sides of the cell. This collection of steps, combined with the testing apparatus, will be referred to as the Multicell Test Sequence (MTS). The testing is conducted by an already-established Supercell that shares a border with the RUT. The Supercell contains the testing apparatus and is responsible for executing the configuration string that represents the raster scan through the RUT and the use of the test triplets for each cell. Note that this Supercell has already been tested and all its cells have been determined to be nonfaulty; thus any pieces of the testing apparatus are nonfaulty. Its access point to the RUT is a wire head that is on the Supercell side of the boundary, and has direct control over a single cell on the edge of the RUT.

A component of the Supercell is a mechanism for transmitting information among a network of tiled Supercells. This was achieved by the implementation of a loop-free broadcast mechanism, called a Link-Lock Network. The basic Link-Lock-capable Supercell is a 16x16 set of Cell Matrix cells, and allows a collection of Supercells to share information among themselves. "Holes" in the tiling caused by faulty regions of the Matrix do not interfere with this information sharing, since information can be transmitted through all four sides of a Supercell. However, each Supercell detects which outputs will create transmission loops, and avoids outputting to those sides. This loop avoidance is important to prevent latches that lock up. Details on this are beyond the scope of this paper and can be found elsewhere[4].

This structure for transmitting signals was then augmented with the necessary structures so that a Supercell can configure new Supercells, on all four sides, in response to incoming configuration information. A collection of Supercells will thus configure, in parallel, a new set of Supercells along its perimeter. The augmented Supercell is a 40x40 set of Cell Matrix cells.

The Supercell circuitry was laid out onto a Cell Matrix cells using the Cell Matrix Layout Editor™ version 2.0[9]. The configuration string needed to create a Supercell was constructed from an automatic configuration generation tool that reads in Cell Matrix Layout Editor files. Sequences of configuration commands were then constructed for configuring a 40x40 region of the Matrix with this Supercell structure. The sequence consists of approximately 37,000 steps. When fed into a single Supercell, this sequence creates a second Supercell. When fed into a 10x10 set of Supercells, the same sequence configures 10 new Supercells along one of the edges of the 10x10 collection. If there are "holes" in the 10x10 tiling, configurations will also occur inside those holes. The sequence requires access to only 6 cells along any edge of the matrix.

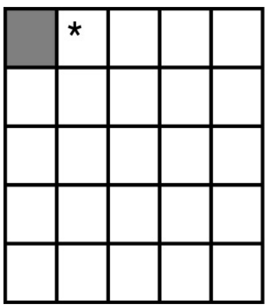
Figure 5a

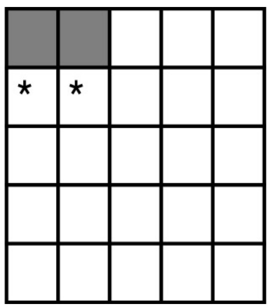
Figure 5b

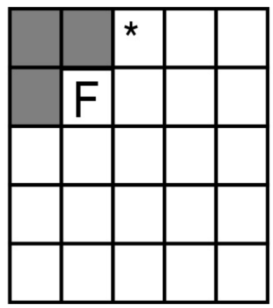
Figure 5c

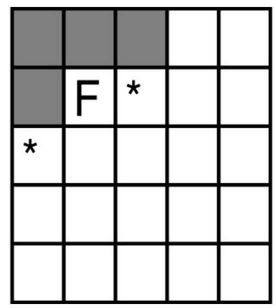
Figure 5d

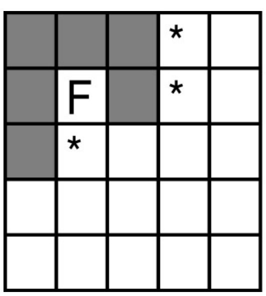
Figure 5e

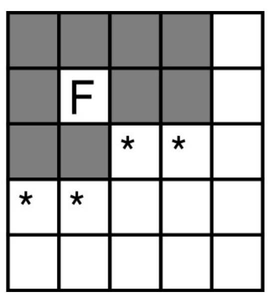
Figure 5f

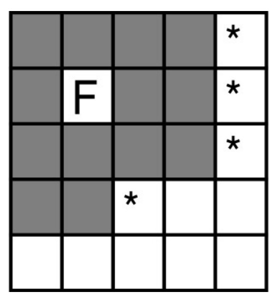
Figure 5g

Once it was verified that these Supercells can be created and connected to a Supercell network, they were augmented to detect and isolate faults within the Cell Matrix hardware. The testing apparatus was added in to the Supercell layout in the Layout Editor, and the configuration string to create these structures was automatically incorporated into the Supercell configuration string. The MTS configuration string was incorporated into the configuration sequence. Fault testing occurs while the Supercells are being configured. Immediately before the configuration sequence is used to build a new set of Supercells, the MTS configuration string is run through the existing network of Supercells. No cells outside the RUT are affected by the test sequences; thus regions which are completely surrounded by Supercells can be fully tested without disturbing the existing network. The test sequence consists of approximately 259,000 steps for a 44x44 test region.

Figures 5a-5g illustrate the results of repeated applications of the MTS. Each square represents a 44x44 region. Grey squares have already been configured as Supercells. Regions marked "*" are candidates for future Supercells, but are presently being tested for faults. White squares are unconfigured. In Figure 5a, a single Supercell has been configured, and is testing a region to its right. In Figure 5b, the tested region has passed, and has been configured as a new Supercell. These two Supercells are now testing regions below them. Note that these two regions are tested simultaneously. If Figure 5c, the region labeled "F" has failed a test, and is now considered faulty. Thus, only one new region is being tested/configured, to the right of the existing Supercell network. In Figure 5d, regions below the existing Supercell network are tested/configured, again in parallel with each other. Figure 5e shows another round of test/configure steps, to the right of the Supercell network. Figure 5f shows another round below the existing network. Finally, Figure 5g shows another round to the right. Note that a continuous block of Supercells (depicted by grey squares in Figure 5g) has been succesfully configured, despite the presence of the faulty region "F" in the middle.

The Supercell design was further enhanced to add fault isolation logic. While the fault detection sequence is running, this isolation logic is periodically armed. It compares expected test patterns with actual return patterns from the cell under test. Differences are detected (after taking into account propagation delays), and cause the setting of a "failure flip flop" along an edge of the Supercell performing the test (the "testing Supercell"). This flip flop activates a guard wall just inside the edge of the testing Supercell. Once activated, this guard wall prevents any cells inside the failed region from accessing cells in the testing Supercell. This effectively isolates the region with the failed cell(s) from the properly-working Supercells. Figure 6 illustrates a guard wall. The cells making up the actual guard wall are all contained in the good region. Cells in the Active Column prevent cells in the Passive Column from entering D mode, by setting their C inputs high. Therefore, cells in the bad region have no access paths to cells in the good region, since only D-mode cells may assert their C outputs[5].

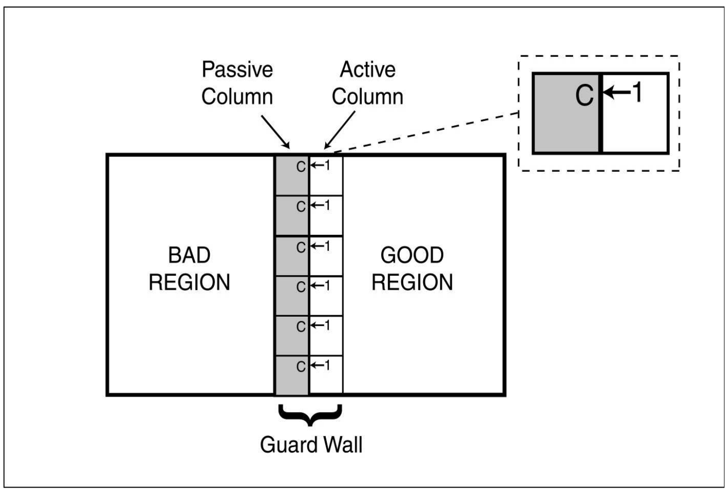

Figure 6. Depiction of a Guard Wall.

Addition of the fault isolation logic increases the size of the Supercell to the final size of 44x44. The size can be easily enlarged if needed by adding negative space in the form of cells with blank truth tables to the Supercell layout.

Figure 7 illustrates the process by which a region containing any faulty cells will be isolated by a ring of guard walls. The Supercell in the upper left corner is repeatedly passed the MTS configuration string, labelled CTA, and this is passed to all Supercells already on the network. The CTA is cycled to be used in 4 directions: East, South, West, and North. The first CTA passed to the network is used by a Supercell to test the RUT on its Eastern edge, which causes it to discover the faulty RUT in the center of the 3x3 matrix. This Supercell activates its guard wall. The next CTA is used by a cell on its Southern edge, to discover the faulty RUT in the center and activate a second guard wall. The third CTA passed to the network is used by a Supercell to test its Western edge, causing it to activate a third guardwall. Isolation of the faulty RUT is completed at the fourth CTA transmission, during which the cell beneath the RUT discovers the faulty RUT on its Northern edge and activates the final guard wall. The faulty RUT has now been completely isolated by surrounding guard walls.

This completes the description of the design and overview of the implementation of the upper tier of the test driver

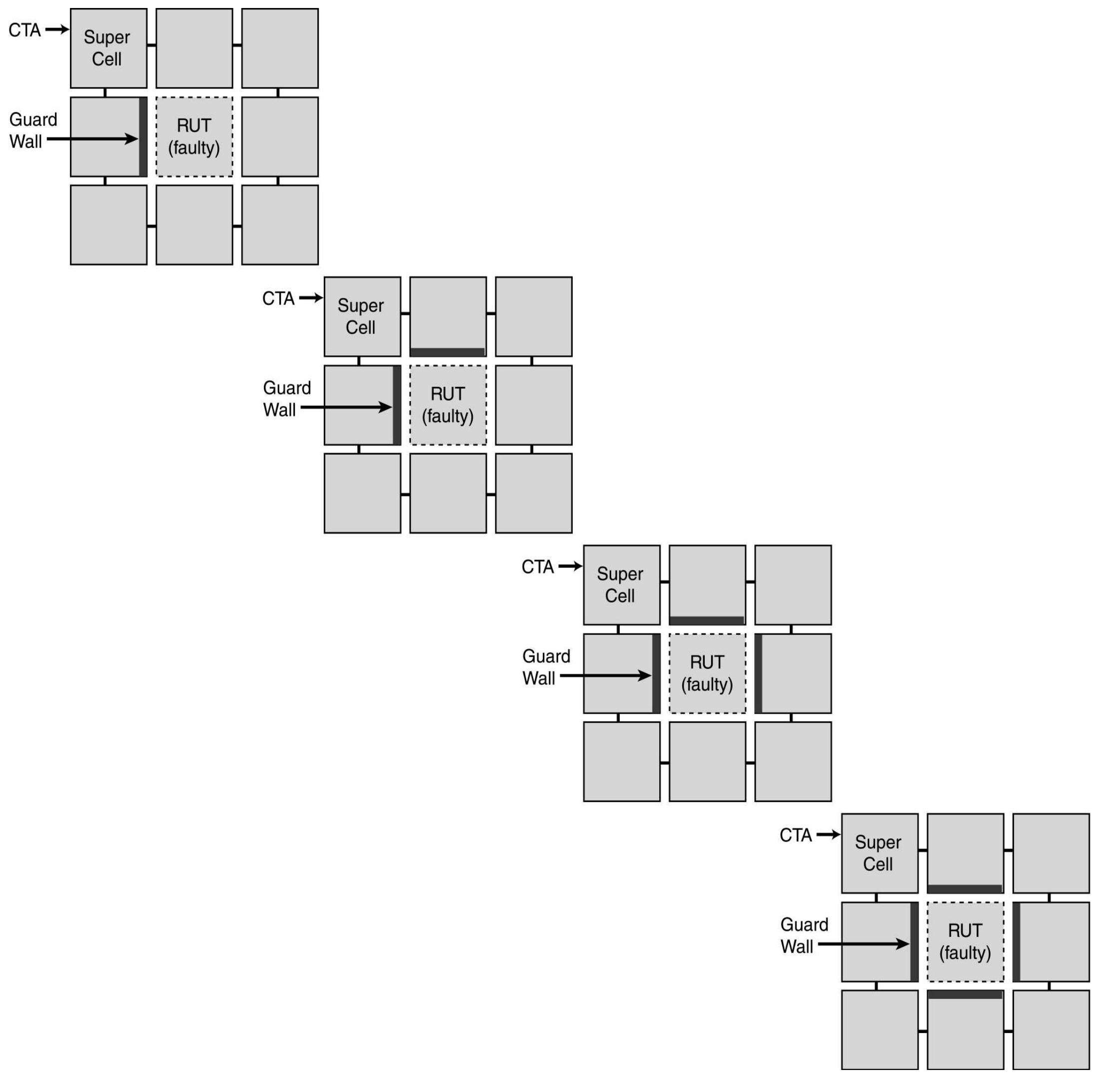

Figure 7. A 4-part time series showing the full activation of guard walls to isolate a faulty region. The faulty RUT is shown in the center.

implemented via Supercells. The whole testing system is implemented as a configuration string that can be passed to Cell Matrix hardware. The Supercell definition is repeatedly passed to the hardware until the hardware is completely tiled with Supercells. At that point all the hardware has been tested for faults, and all regions that the tests find to be faulty are isolated from the rest of the hardware with guard walls that prevent I/O.

The lower-tier and upper-tier test drivers described above were implemented to produce configuration strings. Ideally those strings would then be run on a custom ASIC that implemented an array of Cell Matrix cells, since that is one of the target hardware platforms that could ultimately use the hardware-level fault testing capabilities developed here. However, custom ASIC chips containing a sufficient number of cells for testing this system were not available for this project. As a result, physical, hardware-level faults had to be simulated. Simulated faults were in fact preferable during the development stages of this project because hardware faults could not interact with and complicate the development of the test driver.

Two types of Cell Matrix simulations were used in this research. The first is an all-software simulator[10], which models both the behavior of individual cells as well as the interactions among them. The second is an emulation of the Cell

Matrix architecture on top of a Xilinx XC2S200 Spartan FPGA. For both of these, physical hardware-level faults were also simulated. Cell Matrix configurations were created using the Cell Matrix Layout Editor. Preliminary verification of the Cell Matrix's behavior in the presence of faults has also been obtained using partially-defective 2 micron CMOS prototypes.

3.4 Sample tests

The test sequences access each cell within the test region (RUT), from all four sides (except edges and corners, which access from 3 or 2 sides respectively). When a cell is accessed, its C-mode (configuration) operation is tested, as is its ability to pass and process data correctly. Additionally, all cells within the test region are used to test other cells, and hence are given an additional level of testing. The specific tests performed on each cell can be easily modified for whatever types of failures are deemed to be most likely to occur.

One sample test used was to configure the cell as an inverter and then test the inverter with zero and one inputs. This tests all failure modes as described above: power and clock, configuration circuitry used in C-mode, the lookup table function circuits used in D-mode, and cell input and output lines. It also tests the lookup table for one possible configuration. Output patterns can be analyzed for such things as stuck-at-zero and stuck-at-one problems with the cell I/O lines. The process of configuring the cell to be an inverter will also test for configuration faults, which may be distinguishable from stuck at faults by analysis of the output pattern. If the input line is stuck at zero, the output will be all ones regardless of the input. If the input line is stuck at one, the output will be all zeros regardless of the input. If the input and output are shorted together, then one or the other will prevail and the output will be wrong for one or the other input.

Another sample test was to write and read back the lookup table with an arbitrary pattern. This will test for errors in the configuration mechanism and memory loading, as well as in the I/O lines. Depending on the pattern, this will test for stuck-at-zero or stuck-at-one faults associated with memory locations in the lookup table; for instance, while in C-mode, if a cell is sent a configuration string with alternating zeros and ones, is that string preserved? This test was performed before the test above, because it tested fewer aspects of the cell and could thus be used to better pinpoint the causes of faulty behavior.

4. RESULTS

The ability of the configuration to detect faulty cells was established through experiments with faulty and non-faulty Cell Matrix hardware simulated on the virtual Cell Matrix software program; their traversal of the hardware to include all cell sides was also confirmed through self-checks in the programs that generated the MTS configuration strings, as well as by visual inspection of the printouts from the operation of the configuration string on hardware and by watching the configuration sequence while it ran via a graphical front end.

4.1 Verification of the MTS

The MTS was performed on a very small Cell Matrix emulated on top of a Xilinx XC2S200 Spartan FPGA, using a board obtained from Burch Electronic Designs[11]. On this device, it is possible to implement 64 Cell Matrix cells, and array them in an 8x8 matrix, and to also implement a parallel port decoder, to allow access to any of the 32 sets of inputs along the perimeter.

Using this setup, a small 8x8 collection of Cell Matrix cell circuits was laid out. The first 3 columns of the matrix were used to implement the testing apparatus and other subsections of the Supercell critical for conducting the fault detection sequences on the RUT; the remaining 5 columns and 8 rows represented the RUT. The MTS patterns could be sent to this matrix, which ran the indicated tests, and reported the results. To generate a configuration string for this setup, the program that generates the MTS was rerun with the smaller boundaries, and a new configuration string was constructed. On a fully configured 8x8 Cell Matrix, the configuration string was sent in, and the tests all returned SUCCESS, indicating that the entire matrix was operating perfectly.

Next, the 8x8 array of Cell Matrix cells was re-implemented on the FPGA, but with a single defect in the tiling: a D input to a single Cell Matrix cell was disconnected from its neighbor's output, and instead connected directly to ground, thus simulating a Stuck-At-0 fault that one of the sample tests was intended to catch. The configuration sequence was set up to print out the row, column and cell side it was working on and stop once it found a failed cell. Printed and visual output confirmed that all cells were tested from all sides. The system ran the test and printed out a message corresponding to the correct location for encountering a failure at the sabotaged cell. The configuration string used to implement the testing apparatus from each side of each cell in the RUT was thus able to correctly determine both the existence of a faulty cell, as well as its location within the matrix.

The proper function of the MTS, i.e., individual cell tests using triplets, plus testing apparatus, plus wire-building to access all cells in the RUT, was also independently verified using the software-based Cell Matrix emulator. The proper incorporation of these components into the upper tier (Supercell) was also verified this way.

4.2 Tests of the full upper tier Supercell

The behavior of the Supercell circuit was tested using a Cell Matrix simulator. This simulator models the behavior of a Cell Matrix at the cell level, i.e., by modeling each cell, as well as the interaction among cells. This simulator was enhanced to allow the simulation of faulty cells, e.g., cells which would not respond properly to their inputs. Using this simulator, the ability of Supercells to detect and isolate faults was verified. By varying the pattern of faulty cells, the ability of the system to negotiate complex fault patterns and still function properly was confirmed. Other work[4] goes further than this, subsequently achieving the implementation of a target circuit on the faulty Cell Matrix, working around the walled-off faulty regions while utilizing the non-defective hardware.

The extent of testing was bounded within the region by the treatment of edge cells as edge cases within the program that generated the test access. This prevented a side from being tested that was outside the bounds of the RUT. Testing every cell within the RUT without disturbing anything beyond the region's boundaries was confirmed by the insertion of self-test statements executed in the program that generates the MTS configuration string, as well as by human oversight of printed and graphical output of cell indices and sides from running the test sequence on the Cell Matrix Simulator.

Again using the simulator, with a set of simulated faulty cells, the ability of the Supercell-based test driver to detect and isolate faulty regions was verified. A set of cells on the Cell Matrix edge was chosen as the entry point, and was supplied with the initial Supercell configuration string and used as the entry point thereafter for repetitions of this configuration string. The system functioned correctly for all tested fault patterns, including single isolated failures, collections of localized failures, and concave failure regions. The system also operated perfectly in cases where failures were dispersed, so as to appear at least once in every Supercell row and column. Parallel layout of the Supercells was also observed, as expected.

The end result of this work is a defect-tolerant, self-generating, fine-grained, parallel test driver for accessing individual Cell Matrix cells. Using this approach and implementation, all cells can be tested for faults, and faults are reported at the granularity of single cells, as soon as a particular test fails on one of the sides of the cell. Supercells provide a fault tolerant, distributed, parallel mechanism for testing multiple regions at once, as well as providing a means for the tests to work their way past faulty regions. From any six contiguous accessible cells on any edge of a Cell Matrix, Supercells can be tiled automatically by repetition of the Supercell definition. Faulty regions can be isolated from the rest of the matrix on a per-Supercell basis, with a 44x44-cell granularity. Layout of the Supercells occurs in a wavefront pattern in O(sqrt(n)), and bad regions do not prevent the testing of regions behind themselves in the overall layout pattern.

5. DISCUSSION

The autonomous testing and fault handling strategy presented above is an efficient, robust mechanism for detecting defects in a Cell Matrix substrate. By employing fixed configuration strings, which are independent of the location or nature of faults, external intervention is minimized to the supplying of these fixed strings. The use of previously-tested cells to create the necessary circuitry for subsequent testing helps ensure the success of the method, independent of the pattern or density of the faults in the matrix. The use of parallel test circuits helps achieve an efficient testing of extremely large collections of cells.

The focus of this research is more on the method of delivering and analyzing the results of various test patterns, rather than the selection of those test patterns themselves. The actual test patterns employed in this research were chosen as a simple proof of concept. Ideally, analysis of actual physical defects would reveal the most likely types of faults for a given manufacturing process, and test patterns would be constructed based on that insight. For example, if a cell's truth table is arranged physically as a 16x8 bank of storage bits, it might be useful to use a test pattern such as

0000000011111111000000001111111... Such a pattern would alternate stored bit values from row to row, thus detecting any shorts in the outputs of adjacent rows.

Because the testing methodology described herein employs (for a particular set of test patterns and a fixed device size) a single, unchanging set of configuration strings, it is feasible to test multiple devices in parallel, by sending those fixed strings to all devices in parallel. Note that, despite the fact that all devices would receive identical configuration strings, the final configuration of each device (i.e., the creation of guard walls to isolate faults) will in general vary from device to device, due to the dynamic, adaptive nature of the testing methodology. This is a classic result for algorithms that utilize self-configurability.

It should be noted again that this represents a very fine-grained testing methodology, much finer than the device-level PASS/FAIL which is currently most often used, and much finer than a device-level PASS/FAIL for a given application. In the methodology discussed, 44x44 regions are marked as PASS or FAIL. Still finer granularity is possible, using alternative strategies which do not leave Supercells in place after testing a region.

Finally, it should be noted that the self configurability of the Cell Matrix plays a critical role throughout this testing methodology. The testing methodology must be efficient, autonomous and robust. To efficiently test a large device, the number of locations being simultaneously tested must increase as the testing of the device proceeds. It is also necessary to efficiently build the test circuitry, otherwise the gains of parallel testing will be lost. Thus, already-configured Supercells must assume to task of configuring new Supercells. To operate autonomously, these Supercells must be implemented on the device itself. And to operate robustly, the Supercells must be built from cells that have been previously tested by other Supercells. Each of these criteria thus make intimate use of the self configurability of the Cell Matrix.

6. FUTURE WORK

There are a number of future directions in which this research could be taken. One enhancement would be to report back not simply that a test has failed, but the details of the failure. This would be useful in diagnosing the nature of defects in a device, rather than just the presence of faults.

Another enhancement would be to extend the behavior of the testing subsystem once a fault has been detected. For example, reporting of precise defect locations to a standard place-and-route algorithm (perhaps implemented on the Cell Matrix itself) would allow maximal usage of defect-free regions in the matrix, while still avoiding faulty cells. Such reporting and collecting of defect information would require arbitration among multiple Supercells that simultaneously detect defects in their regions under test. Other work[4] describes a distributed way to utilize this fault location information, without gathering such information in a centralized location, and without dependence on a centralized controller for place-and-route.

We would like to use this testing methodology to examine physical devices, and analyze their real-world fault patterns. This could also be extended to testing the behavior of chips in hazardous environments such as those with high radiation or extreme temperatures. The testing methodology presented in this paper could be used to study the nature of the faults which occur under such conditions.

In practice, the Supercell size can grow beyond the 44x44 cell structure used for simple fault testing. For larger Supercells, test failures result in larger regions of cells being marked bad. However, it should be possible to re-capture some of those larger areas, by shifting the placement of a Supercell (so as to avoid a defect), rather than marking an entire Supercell-sized region as bad.

More intensive testing could be done on actual hardware than on the simulated Cell Matrix used for most of this work. Since the test process is highly parallel (and its parallelism increases the longer it runs), the time penalty of running a simulation on a single CPU becomes increasingly great. Therefore, we would like to continue this research using a large physical Cell Matrix.

This test methodology extends quite naturally to a three dimensional Cell Matrix. We would thus like to explore that area of fault testing. In particular, since routing is in some sense "easier" in three dimensions than in two, we expect than the presence of faults will be less disruptive to testing on the three dimensional Cell Matrix than on a two dimensional one.

We would also like to work with a semiconductor manufacturer to apply this research as a process driver[8], i.e., as a way to debug the semiconductor fabrication process itself. Since gross defects in the fabrication may still allow a Cell Matrix to operate properly, and since such a matrix can then be used to investigate the location and nature of defects, this could be a way to efficiently analyze a fabrication process.

ACKNOWLEDGMENTS

This work was funded in part by NASA SBIR Contract NAS2-01049. The authors gratefully acknowledge the support of NASA and Ames Research Center in the performance of this research.

REFERENCES

1. Ralph C. Merkle, *Nanotechnology is Coming*, http://www.merkle.com/papers/FAZ000911.html, September 2000.
2. Nicolas Mokhoff, *High cost of test scares off investors, conference goers told*, EE Times Story, October 31, 2001.
3. Xilinx, Inc., *Xilinx Introduces Breakthrough Cost Redunction Technology for Virtex-II FPGA Family*, Press Release, March 25, 2002.
4. N. J. Macias and L. J. K. Durbeck, "Self-Assembling Circuits with Autonomous Fault Handling," to appear in *Proceedings of the Fourth NASA/DoD Workshop on Evolvable Hardware*, July 2002.
5. US Patent #5,886,537
6. N. Macias, "The PIG Paradigm: The Design and Use of a Massively Parallel Fine Grained Self-Reconfigurable Infinitely Scalable Architecture," *Proceedings of The First NASA/DOD Workshop on Evolvable Hardware (EH'99)*, D. Keymeulen, J. Lohn and A. Stoica, eds., pgs. 175-180, July 1999.
7. US Patent #6,297,667
8. L. J. K. Durbeck and N. J. Macias, "A Process Driver for Nanofabrication: Detecting and Analysing Hardware Defects using the Cell Matrix Computing Architecture," submitted abstract for *The Ninth Foresight Conference on Molecular Nanotechnology*, available from http://www.foresight.org/Conferences/MNT9/Abstracts/Macias/index.html, November 2001.
9. http://www.cellmatrix.com/entryway/products/software/layoutEditor.html
10. http://www.cellmatrix.com/entryway/products/software/simulator.html
11. http://www.BurchED.com.au