Adaptive Methods for Growing Electronic Circuits on an Imperfect Synthetic Matrix

N. J. Macias[+] and L. J. K. Durbeck[*]
Cell Matrix Corporation

RUNNING HEADLINE: Adaptive Circuit Growth on an Imperfect Matrix
KEYWORDS: Self Organizing, Cell Matrix, Adaptive, Nanotechnology, Fault Tolerance

ABSTRACT

Living systems can adapt to injuries and even heal themselves, an ability desirable also in synthetic systems. A method is presented for dynamically adapting the construction of an electronic circuit to hardware defects by formulating the process as a series of interactions between identical but specialized structures called supercells. The circuit components, including wires, can occupy any place in the hardware that has been determined to be free of defects. The circuit specification is reduced to a connected graph, with no positional information, and provided as a code repeated in each supercell. Using the code, supercells differentiate into circuit components in a late stage of the process, with with highly adaptable physical location and organization; supercells also form the wires between circuit components. The structure and function of the system at three major levels is presented, the lowest cellular level, the supercell, and the target circuit level. Adaptation of circuit construction to defective hardware was observed for this method. Results obtained from this development process on simulated and real hardware with a variety of defect types and defect patterns are presented, as well as higher level simulations of the algorithm and its response to a wider range of defect patterns, amounts of hardware, and hardware to fault ratios.

N.J. Macias, L.J.K. Durbeck/Biosystems 73 (2004) 173-204 ©2004 Elsevier Ireland Ltd.

1. INTRODUCTION

There has been ongoing effort by researchers to invest electronic circuits and electronic systems with the ability to survive physical or electrical damage to their electronic innards. Electronic circuits are in widespread use; however, nearly all circuits in use today cease to function properly when they incur any sort of damage. In many cases the tradeoffs between system size and complexity will continue to be such that it is cheaper to create a compact, nonrobust implementation of an electronic component and then replace it when it fails; still, there are cases where it is too expensive to access the system – such as a system on a probe launched into space – as well as cases where it is too expensive to communicate with the system to remotely diagnose its failure state and repair it – such as a system sent into deep space, beyond the casting limits of rich communication links. And it appears that as the trend toward ever more complex systems continues, it is also becoming desirable to design the subcomponents of complex systems such that they diagnose and repair themselves, simply to reduce the complexity of the larger system that is controlling and coordinating everything, and also to reduce the amount of time the system spends running diagnostics (by having the subcomponent self-diagnostics run in parallel).

The great majority of digital electronic circuits in use today are implemented in a compact, fairly direct representation of some kind. The most compact form in common use is representation of each logic gate using transistors that are created out of semiconducting materials, and wires built of metal, using a technique such as custom ASIC. This method is extremely compact, but any error in the creation of the transistors or wires is generally fatal, and the electronic circuit will generally fail. A technique such as custom ASIC can be viewed as a literal but imperfect transcription of the circuit design, or definition. However, the abilities to detect, diagnose, and overcome defects can be incorporated into the design of the circuit and its operation.

There are electronic circuits with structures that permit a good degree of tolerance to custom ASIC fabrication errors. Programmable logic devices are a good option, because rather than implementing the desired electronic circuit directly in silicon, a more general circuit that contains many programmable logic elements is built in silicon, and that circuit is then programmed – a step completely separate from fabrication – to implement the desired circuit. Researchers have been developing techniques and tools to alter the configuration step so that it first looks for defective logic blocks, and then avoids using them in the creation of the desired circuit.

Our approach falls into this category. It is illustrated in Figure 1.1 and 1.2. In silicon we construct a multidimensional array of identical programmable elements called cells. We later program that array of cells. In this paper we describe a technique for programming that array so that the first thing that each region of programmed hardware does is look at the regions immediately surrounding it and find any faults. If it finds faults, it walls itself off from any defective region of hardware around it so that the faults cannot spread to itself. After that, it sends out signals to neighboring hardware, and the neighbors do the same, forming a fully connected communication network. Using that communication network, the regions of hardware send out signals to tell the other regions what piece of the desired circuit they have elected to implement. They follow a prescribed set of behaviors so that this step is well-coordinated, and only one copy of each circuit component is built. They also keep those parts of the communication network that mimic the wires between the components of the desired electronic circuit. They then signal to the external world that they are done setting up the hardware, and from that point in time, they perform the function of the desired circuit.

What is unusual about this approach is that it permits the hardware to test and set up itself. It does this by introducing an extra step between fabrication of the hardware and programming the hardware to implement the circuit – namely, programming the hardware with an intermediate structure and function whose main purpose is to allow regions of hardware to, separately, locally, and in parallel, perform hardware tests, and then to allow regions of hardware to communicate and act collectively to set up the desired circuit.

Benefits to be had from this overall approach are several. The result is a piece of hardware that contains a robust version of the digital electronic circuit – the electronic circuit works perfectly even if the hardware contains a lot of defects. The lowest level of the system, the physical hardware constructed from transistors or switches, is unusually robust in that it is comprised solely of identical components that are simple, small, and interchangeable. This provides a high degree of redundancy at the lowest level of the system, redundancy that can permit many options for the placement and interconnection paths between the components of the desired circuit. Even the wires, or signal routing, of the desired circuit are programmed in, and can therefore be programmed in after diagnosis of defects.

The question we were most interested in asking and answering was whether and how to introduce higher order behavior into the system without losing features of the physical cell level that we think are critical. The hardware-level cells of a Cell Matrix are extremely simple electronic components, and they have several features that permit them to perform simple electronic functions and to communicate with neighboring cells, either to exchange information, or to modify each other's simple behavior. Features we were interested in bringing along to the higher level of the system include the low degree of inherent individuality of cells, the high degree of fault isolation due to their physical connections only to nearest neighbors, and cell self-duality, which is the ability for all cells to interpret incoming information as data, or as code to change its internal behavior, and thus to act either as a subject or an object of reprogramming (programmer or target). More complex behavior is achieved by programming a group of adjacent cells to implement a larger entity. It was our object to design a multicellular program that exhibited higher order behaviors but was still able to do all the things a single cell can do.

The supercell described here is such a multicellular program, and its higher order behaviors include the ability to interrogate a neighboring region for a wide variety of faulty behaviors, the ability to create new supercells, the ability to create communication

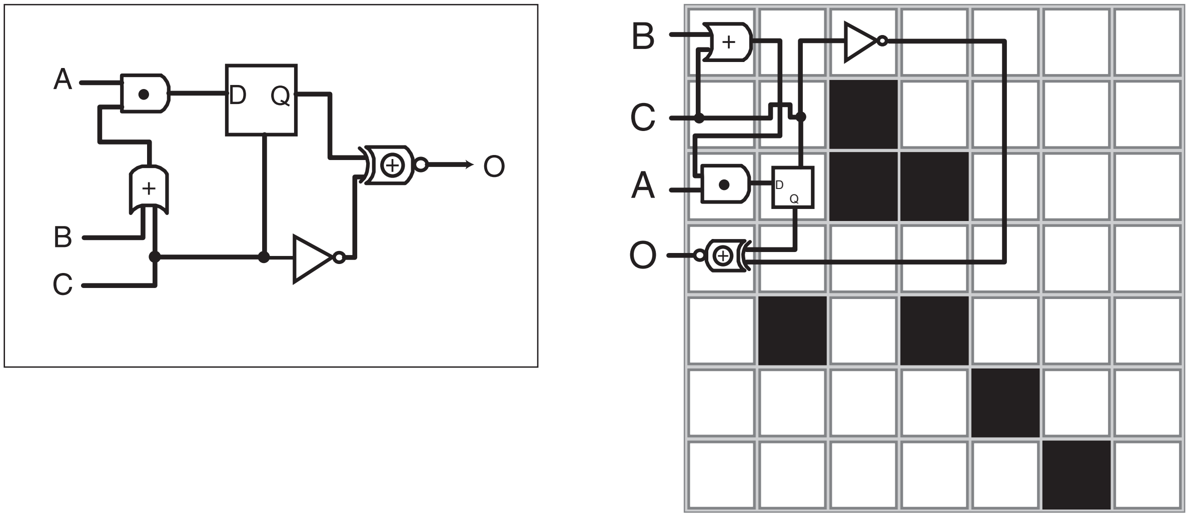


Figure 1.1
Laying out an Electronic Circuit on Defective Hardware

The schematic diagram on the left specifies the design of a simple circuit. Extremely compact hardware implementations of this circuit are possible using custom ASIC or programmable logic devices; however, such representations are not at all robust to hardware defects or failures. The diagram on the right shows how the circuit can be implemented on top of a large array of Cell Matrix cells that has been configured as a set of supercells. The circuit coexists on the hardware along with defective cells, but it uses no defective cells in its implementation, so that it can function perfectly. The technique can also be repeated on the same piece of hardware to counter new defects or damage: it will lay the circuit out differently so that it continues to function perfectly.

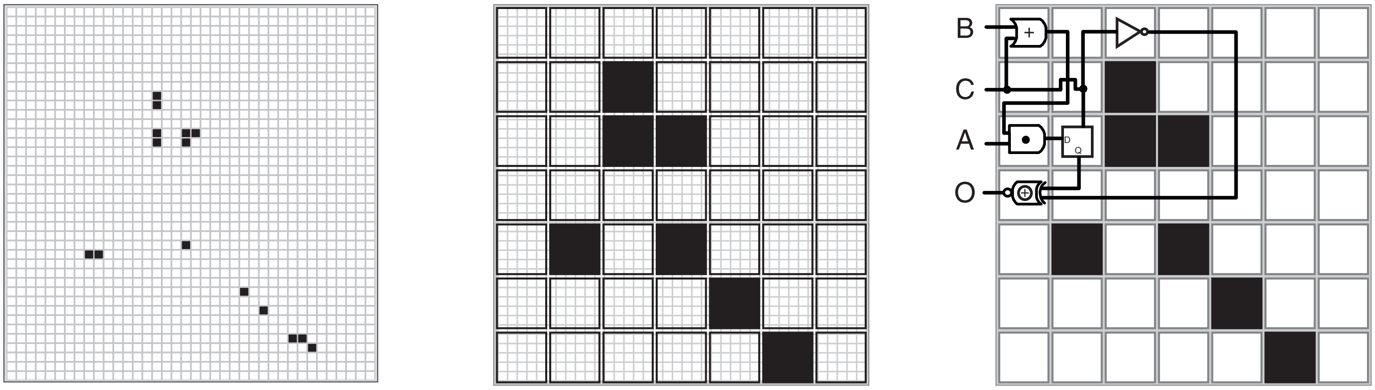


Figure 1.2
Basic Approach

The three main stages of the approach are illustrated left to right. The process begins with a piece of Cell Matrix hardware containing an arbitrary number of cells, shown as small grey boxes, and a random distribution of defects, shown as black boxes. A sequence of configurations strings provides the hardware with both static, structural configuration information, and dynamically interpreted instructions. As shown in the middle graphic, this sequence tiles the hardware with a repeated configuration called a supercell, which occupies a 270 cell-square region. In regions where defects are detected, no supercells are created, as shown by black boxes. After the matrix is tiled with supercells, the supercells form a communication network and then differentiate into the various components of the desired final circuit. They also establish permanent links that correspond with the wires in the desired circuit. The rightmost drawing shows the final result, a network of supercells that implements a small circuit. The circuit is automatically laid out on the damaged hardware, and able to run perfectly despite the presence of defective cells.

networks within the set of supercells, the ability to query one another and cooperatively build up maps or routes to tell one supercell where another supercell is located, the ability to differentiate into circuit elements, the incorporation of state machines and the ability to form collective state machines, providing them a specific temporal behavior of performing a specific sequence of operations. A Cell Matrix tiled with supercells is illustrated in Figure 1.3.

Supercells retain the desired characteristics of single cells. All supercells start off identical, providing the higher level system with a low degree of individuality of its components. The tiling of supercells achieves the subject/object relationship that cells have: supercells create other supercells, and they do so by passing the supercell definition they receive as data to the target region. Faults are contained at the same granularity as supercells, and this is achieved by each of the supercells adjacent to a region containing a defect turning on their guard walls.

BACKGROUND

Digital circuitry is traditionally extremely fault sensitive. Typically, every transistor within a circuit plays a critical role, and, should a single transistor fail, the entire system will likely fail to some degree. A few exceptions exist. For example, high-density memory contains redundant bits that are used to detect errors in the other bits, and can also be used to correct them (Vanstone and van Oorschot, 1989). At a larger scale, some systems achieve fault tolerance via component-level redundancy. For example, the Space Shuttle currently employs five copies of its on-board computer systems (NASDA 1996). In such a system, critical elements are replicated two or more times, so that should an element fail, a replacement can be switched in to take its place. These systems are extremely reliable in the case of single fault events, but are limited in the severity of damage they can incur, depending on how many redundant copies of critical elements are available. Generally, the more redundancy a system has available, the greater the degree of damage it can sustain.

A key factor that limits the number of available redundant elements is simply the size and complexity of those elements. If a system contains a large number of critical elements, and it keeps three copies of each, this triples the size of the system. Yet the fault detection system could still be disabled by just one fault, and the entire system could malfunction following just two faults, should they occur in any two copies of a single element.

Clearly, this problem is compounded if the elements in question are large: it becomes impractical to maintain, say , 1000 copies for the sake of redundancy.

If, however, the elements of a system are extremely simple and highly interchangeable, then you may be able to keep a huge number of redundant copies within your system without significantly increasing the size of the system. For example, a system containing 1,000 elements that are identical could theoretically have 1,000 spare copies available by

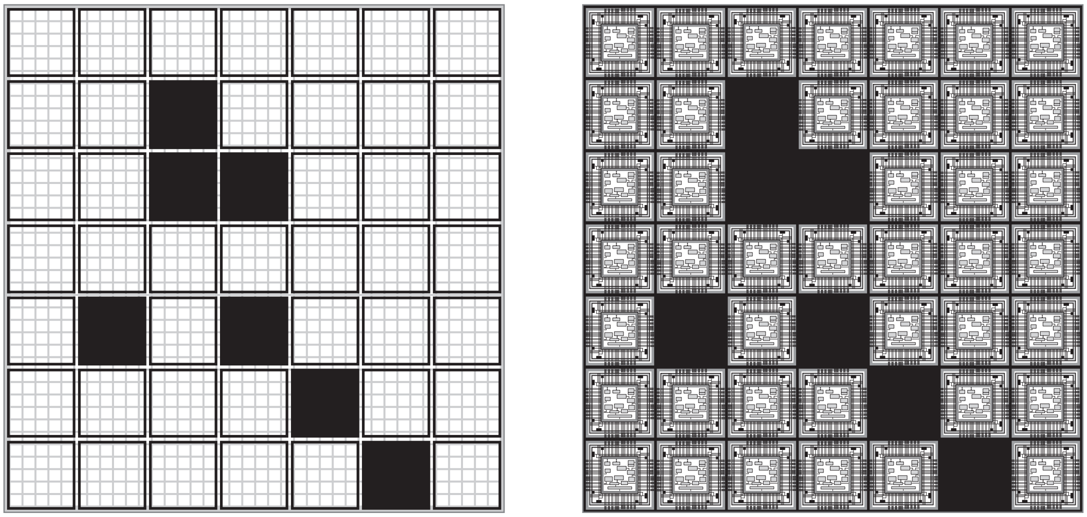


Figure 1.3
Defective Matrix Containing Supercells, Increased Detail

The middle drawing from Figure 1.2 is shown on the left, with the contents of the non-black boxes shown in greater detail on the right. Black boxes are regions containing faulty cells; white boxes are supercells. Black boxes: in each region where a faulty cell is detected, the region is not set up; further, the regions around it guard themselves from it, effectively rendering it as a hole in the matrix. Supercells: Each of the non-black boxes on the left represents a region that has been set up to contain a supercell, and on the right, the structure and contents of a supercell are displayed in greater detail. A supercell uses up a region containing 270 x 270 Cell Matrix cells; each cell in the region is set up, via configuration of its internal memory, to implement some small part of the functions needed for the supercell to operate.

simply doubling the size of the system, since any of those 1,000 spare copies could be used to replace any of the 1,000 identical system elements.

PROGRAMMABLE LOGIC

This observation leads to some promising work in the field of Programmable Logic (OptiMagic 1997). In a programmable logic system, the underlying hardware of the system does not directly implement the system's functional behavior. Rather, the underlying hardware is a substrate whose reconfigurable elements can be configured to perform certain specific tasks. Thus, there is not a particular piece of hardware which, say, acts as an adder. Instead, a set of reconfigurable elements is configured to act as an adder. The advantage of this approach is that, should the elements associated with that adder become damaged, the role of the adder can simply be transferred to another set of reconfigurable elements. Thus, the system has extremely high redundancy, on the order of the number of unused reconfigurable elements in the system. The classic example of this approach is the Teramac work of Hewlett-Packard Laboratories (Amerson et al., 1995; Heath et al., 1998). This reconfigurable-logic based computing system is first analyzed to locate defective components, that are then avoided by the compiler when implementing a design on the system. The result is a highly fault-tolerant system that can function despite having numerous defects.

Despite the success of programmable logic-based fault tolerant systems, many such systems share the common attribute of being externally controlled. In other words, an outside system must analyze the hardware, determine the location of the faulty elements, decide how to implement the target circuit using the undamaged hardware, and then perform the configuration of the substrate (Culbertson et al., 1996). Such a setup, using an external analysis and control system, may work well in the laboratory. However, it has serious shortcomings if, for example, your analysis system is sitting on earth, while your target circuit is sitting on the surface of Pluto. In such a case, the analysis and re-configuration of the target circuit must be done across 3.6 billion miles of space. The communication time alone (over 10 hours round-trip at the speed of light) may make this externally-controlled system impractical.

In such cases where autonomous operation is required, a biologically-inspired approach may be more appropriate (Mange 2000b; Bradley 2000; Ortega-Sanchez 2000; Prodan 2001; Stauffer 2001; POE 2002). While some of these approaches exhibit a high degree of autonomous fault tolerance, many of them use specialized hardware, and many are limited in the patterns of faults they can tolerate. For example, some of these approaches, such as row and column replacement, make assumptions about how many faults will occur in a certain region (row or column), and thus may have difficulty recovering from a fault in every row or column, even if there is still theoretically enough undamaged hardware to re-implement a working circuit. Others require some degree of cooperation from failed elements, for example, a failed element may be required to route signals from one side of itself to another. In the present work, a very general-purpose platform is employed, without any modification to its underlying architecture. The present work allows a wide-range of fault patterns; as long as a sufficiently large connected set of cells is available, the circuit will be successfully rebuilt despite faults. And, very few assumptions are made regarding cooperation from failed system elements; the behavior of just the correctly-functioning elements is sufficient to rebuild a failed circuit. Guard Walls, which are used to isolate faulty hardware regions, are activated in **non-faulty** supercells only. No part of a failed supercell participates in the construction or functioning of the final target circuit. Signals are never routed through faulty regions or failed supercells. Note that in exchange for these additional features, the present work consumes a large amount of hardware, creating an extremely complex atomic unit.

More general background may be found in Jon von Neumann's work on self-replicating automata (von Neumann 1966) and in Burks' work on growing automata (Burks 1961).

THE CELL MATRIX

The reconfigurable platform used in this research is called the Cell Matrix^{TM}. The Cell Matrix is an extremely general purpose reconfigurable platform, with capabilities that go beyond those of other reconfigurable platforms (Macias 1999; Durbeck 2001a). Cell Matrices also possess a number of features conducive to this type of fault tolerant work:

- the architecture of a Cell Matrix is extremely simple;
- the architecture is perfectly scalable, as elements are connected only to immediately adjacent neighbors in a perfectly regular tiling;
- all of the matrix's elements are identical to each other, thus easing the manufacturing burden;
- the architecture possesses a natural fault isolation feature, in that defects in one region of the matrix generally have no effect beyond a very small perimeter around the defect (Durbeck 2002); and
- all of the elements in a Cell Matrix are interchangeable.

Cell Matrices have been studied extensively, both in terms of their inherent properties and their suitability to a variety of applications (Cell Matrix Corporation 1999). Much work has been done in understanding how to manipulate and control Cell Matrices. Much of this work has been done using simulators (Cell Matrix Corporation 2000), though some work has also been performed on silicon prototypes. There is however nothing inherently silicon-based or electronic about a Cell Matrix, and future implementations are envisioned in whatever manufacturing technologies are best suited to its unique characteristics (Durbeck 2001b). Currently, there are a number of development tools being used to implement circuitry on the Cell Matrix, including a graphical layout tool (Cell Matrix Corporation 2001). Other tools and development environments are currently under development.

The present work may be viewed as an example of Embryonics (Mange 2000b; Jackson 2001; Mange 2000a). However, it was not intended to be an embryonics project *per se*, nor was it intended as an example of bio-inspired computing. Rather, it was intended to show the more general technique of creating larger-scale cells (“Supercells”) from simpler building blocks (Cell Matrix cells), in a way that enhances the functionality of the underlying cells without disturbing their essential critical characteristics.

2. APPROACH

CELL MATRIX BACKGROUD

The atomic unit of a Cell Matrix, called a cell, is a simple information processing unit. A Cell Matrix consists of a collection of cells, arranged in a fixed, regular tiling, with adjacent cells connected to each other according to a fixed, pre-defined interconnection scheme. Each cell receives information from its fixed set of neighboring cells, along a set of “D” inputs, processes that information, and generates new information which is sent back to those neighbors on a set of “D” outputs. The manner in which incoming information is processed by a cell is determined by a small program located inside that cell. A cell's program can direct the cell to perform any logical operation on the incoming data. Thus, a cell can produce simple logical combinations of information (AND, NOR, etc.), simple arithmetic combinations (ADD, SUB), decision operations (IF A THEN B ELSE C), and so on. A cell can also be used simply to pass information from a second cell to a third. Figure 2.1 illustrates sample configurations of a single cell. Figure 2.2 illustrates a 2-D tiling of cells in a Cell Matrix.

The behavior of the Cell Matrix is determined solely by the configuration of its constituent cells. Since the interconnection among cells is fixed throughout the matrix, it is only the collective programming of each individual cell that determines the behavior of the entire matrix. By appropriately configuring a set of cells, they can be made to act in concert to perform various higher-level functions, just as in today's silicon circuitry, individual transistors are combined to create integrated circuits with specific desired functions. Figure 2.3 shows a set of four cells acting together to implement a four-bit adder. Each cell acts as a simple one-bit full adder, and these are combined in the standard way to implement a ripple-carry adder.

Therefore, the Cell Matrix is capable of implementing essentially any computational circuit, simply by configuring cells in an appropriate fashion. Such a circuit performs data processing by receiving incoming data, processing it via the collection of cells, and generating outputs. However, the Cell Matrix has additional capabilities beyond those of traditional data processing, discussed next.

SELF-DUALITY

Because the Cell Matrix is composed only of cells, without any additional system components, its behavior derives entirely from the characteristics of its cells. One of the most important features of a cell is its ability to interpret incoming information in not one but *two* possible ways: either as code or as data. This is achieved by having each cell operate in one of two modes: D-mode and C-mode. When a cell is in D-mode, it

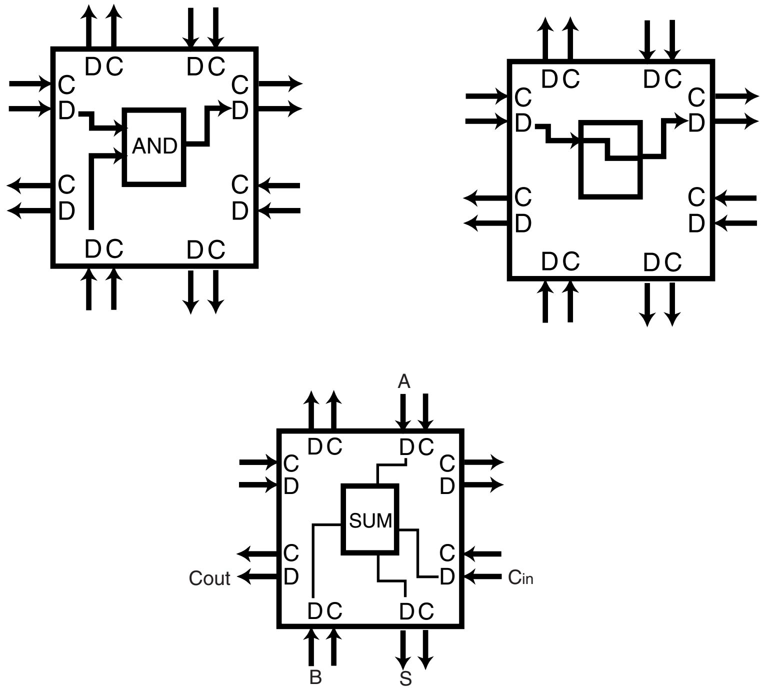


Figure 2.1
Sample Configurations of a Cell
The box in the middle of a cell represents its internal program.
A single cell can be configured as, for example,
an AND gate (top left), a one-bit adder (bottom), or
even a simple wire (top right).

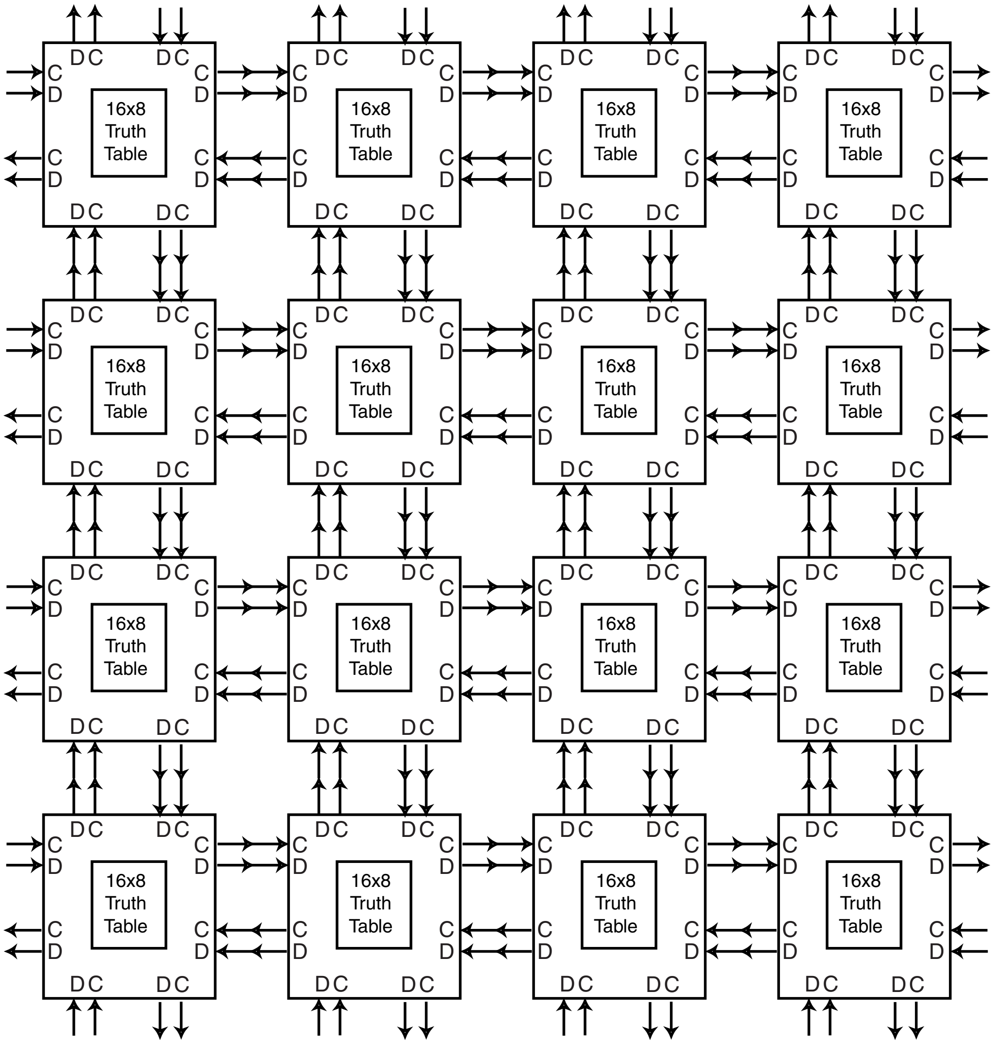


Figure 2.2
Sample Two-Dimensional Tiling of
Cell Matrix Cells

Each of the 16 rectangles represents a single cell.
This tiling is independent of the size of the matrix.
If four of the above 4x4 matrices were tiled in a 2x2 pattern,
the result would be an 8x8 matrix.
Each cell contains a 128-bit memory, organized as a 128-bit truth table.
This truth table maps each of a cell's 16 possible 4-bit D-input combinations to
the cell's 8 output values (4 D and 4 C).

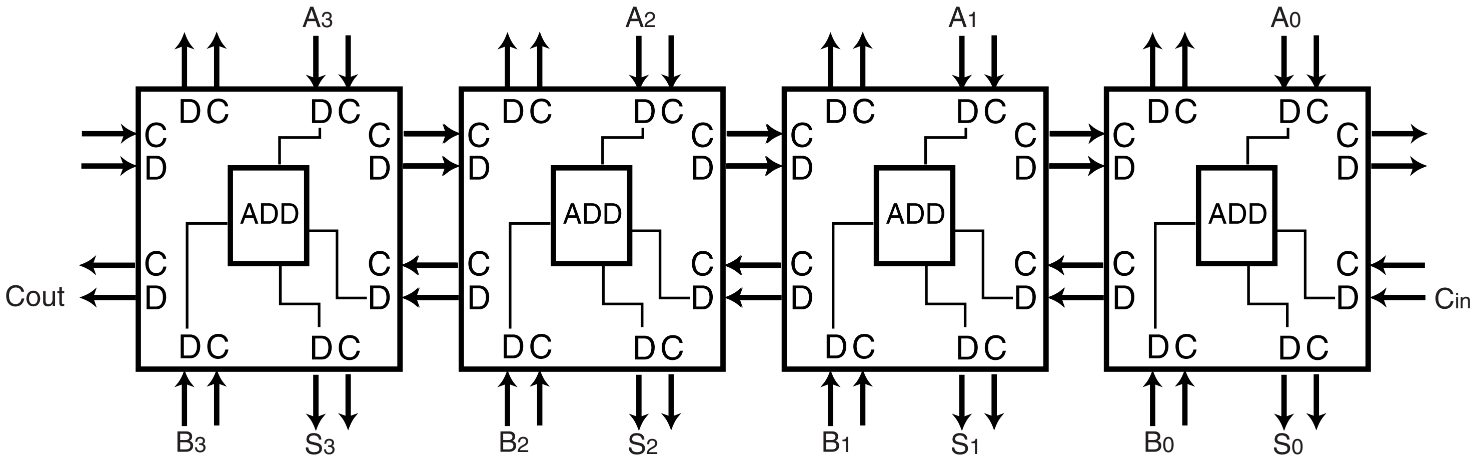


Figure 2.3
Four cells acting together to implement a four-bit adder

Cin is the initial carry in. Circuit adds
[A3 A2 A1 A0] and [B3 B2 B1 B0] and produces
[S3 S2 S1 S0] and final Carry out.

interprets incoming bits as data, and processes those bits with its internal program. When a cell is in C-mode, it instead interprets incoming bits as code, using them to rewrite its internal program. The mode of a cell is determined by its C inputs: if any C inputs are set to 1, then the cell is in C-mode; otherwise, the cell is in D-mode. Therefore, a cell's mode is determined by its neighbors. Figure 2.4 illustrates this difference between D-mode and C-mode.

The ability of a cell to interpret an incoming stream of bits in one of two ways is referred to as *self-duality*. Figure 2.5 illustrates this notion in a sequence of three steps. In the first step, at the top of the figure, three cells, X Y and Z, are each operating in D-mode, and thus are exchanging data with each other. Their internal programs (labeled “16x8 Truth Table”) are static. In the second step of the sequence (middle of Figure 8), cell X is asserting a 1 to one of cell Y's C inputs, thereby placing cell Y into C-mode. In this configuration, X's D output is being used to modify Y's internal program. X could also read Y's prior program via Y's D output. In the third step of the sequence (bottom of page), cell Y has returned to D-mode, and is now configuring cell Z by asserting a 1 to one of Z's C inputs. Thus, Y may alternately play the role of being an object (second step) or a subject (third step) of a configuration operation. This interchangeability is the feature to which “self-duality” refers.

Self-duality is a powerful cell-level feature that allows the creation of autonomous, dynamic systems within the Cell Matrix, including systems that self-replicate or otherwise change and grow while running (Macias 1999). In the present work, self-duality is used in many ways, but most notably to allow structures (supercells) to be created, and then subsequently used to create further supercells. This dual-operation of supercells is key to minimizing external intervention in the fault testing and circuit assembly phases of the system. Rather than relying on external mechanisms, the necessary mechanisms are built within the system itself, using supercells to both implement the mechanisms themselves, and to create the supercells that are used to implement the mechanisms. The task of fault avoidance then becomes one of ensuring that supercells occupy only fault-free regions of the hardware.

Despite having the important property of self-duality, cells are actually very fine-grained atomic units, and by themselves are capable of performing only very simple operations. This simplicity of a single cell is a deliberate feature, and has important implications for manufacturing (Durbeck 2001b). However, it has the consequence that even most simple applications require a large number of cells, all working together to implement a higher-order function.

PASSAGE FROM CELLS TO SUPERCELLS

It is often difficult to design complex systems directly from the simplest, lowest-level atomic units. Suppose a complex system is to be implemented on the Cell Matrix. Rather than directly designing it from individual cells (which, remember, are extremely simple), one may instead first implement larger, more powerful atomic units, called supercells. The final complex system is then designed using supercells as an atomic unit. This may

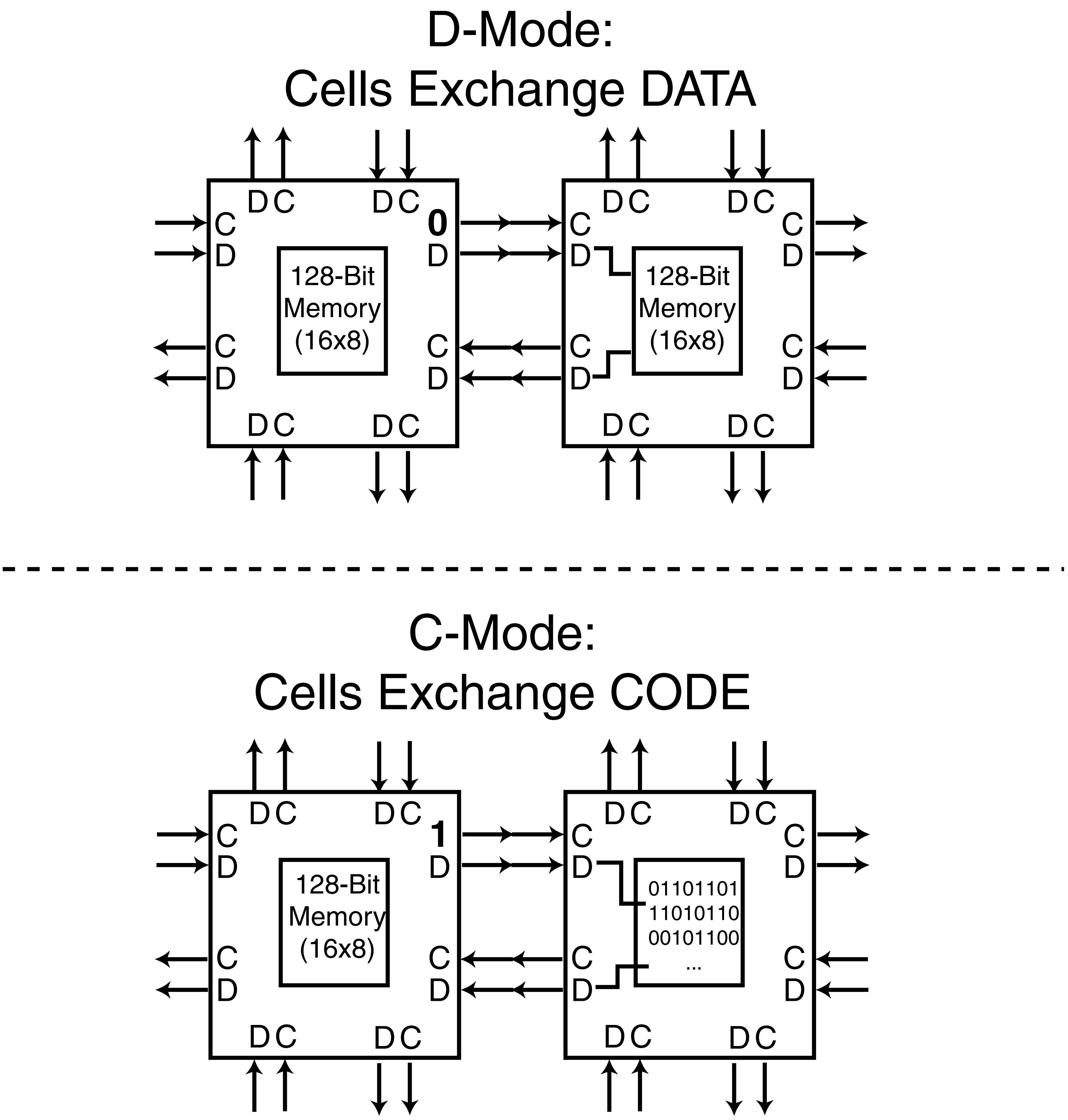
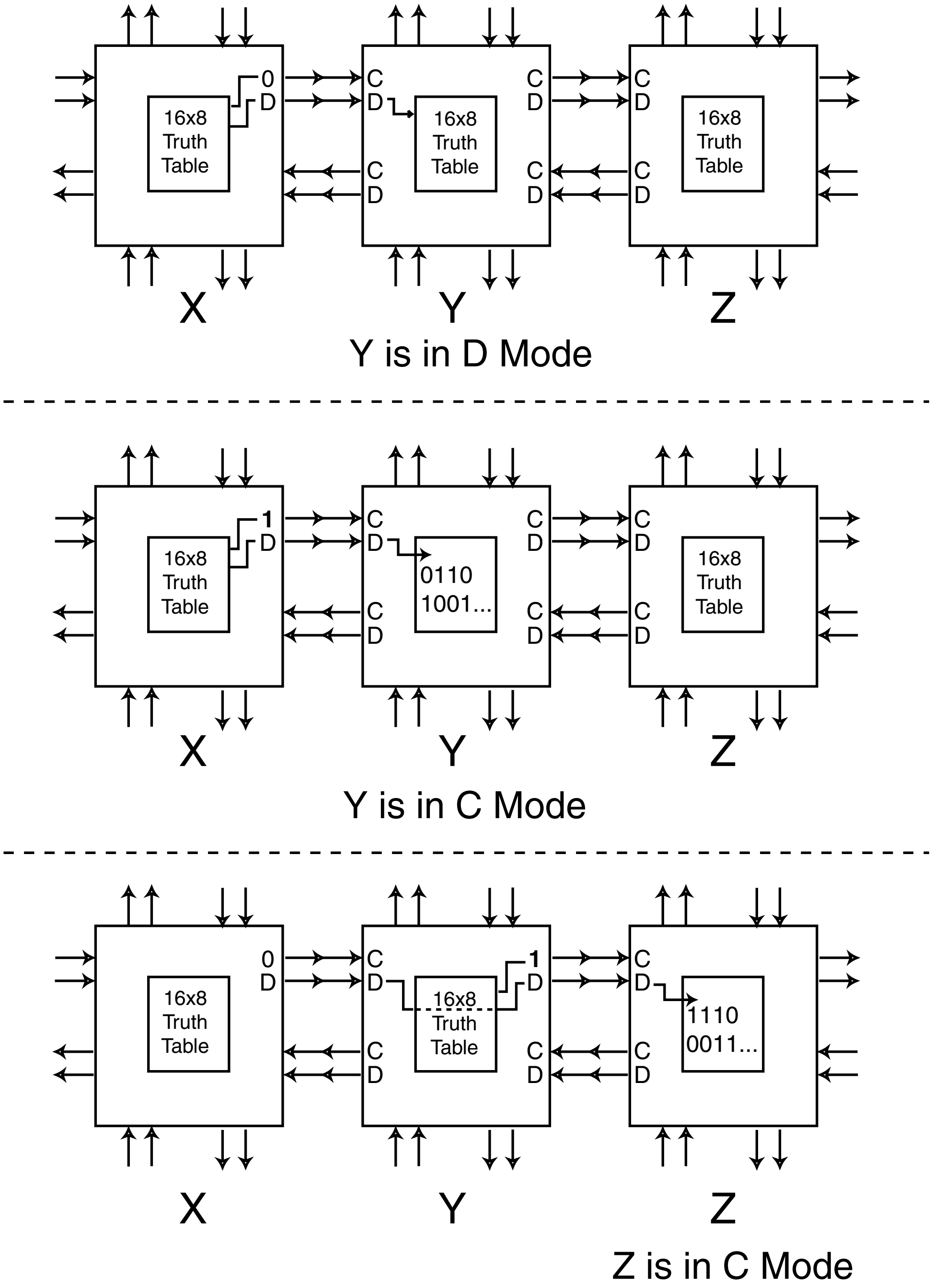


Figure 2.4

D- and C-mode behavior of a cell

If a cell's C inputs are all 0, then the cell is in D-mode, and incoming
D inputs are processed by the cell's program.
If any C inputs are 1, then the cell is in C-mode, and incoming
D inputs are used to re-write the cell's program.
The cell's D outputs present the old program.



Y is in D Mode

Y is in C Mode

Z is in C Mode

Figure 2.5
Sequence Illustrating Self-Duality in Cells
In the top figure, all three cells are in D-mode.
In the middle figure, cell X has placed cell Y into C-mode,
by setting Y's Western C input to 1.
Cell Y is thus being reconfigured, i.e., it's internal program is being changed.
In the bottom figure, cell Y is again in D-mode, and is now itself changing
cell Z's configuration, having placed cell Z into C-mode.
This interchangability of roles, from object to subject, is a key
to creating autonomous, dynamic systems on a Cell Matrix.

greatly simplify the design process, particularly in cases where the design of the system is being performed by an automated system, as opposed to a human designer.

What characteristics should a supercell possess? Having noted the importance of self-duality at the single-cell level, it is reasonable to believe that self-duality may also be important at the supercell level. One goal of this work was thus to design a supercell which possesses an analog of a cell's dual-interpretation of incoming information. Before describing this supercell-level self-duality, we must first discuss how supercells come to be.

The only mechanism for changing a cell's internal program is to first place the cell in C-mode, then send a program into the cell, and then place the cell in D-mode. For cells whose inputs are available from outside the Cell Matrix, this is simple to accomplish. Most cells, however, have all of their inputs connected to other cells within the matrix, and thus those inputs cannot directly be accessed from outside the matrix. To change the programming of such cells, one must use neighboring cells that **do** have access to the target cell's inputs. Therefore, the process of configuring a cell generally consists of a series of steps, during which cells are first themselves configured, and subsequently used to configure other cells. This sequence of operations relies heavily upon the self-duality of cells (Macias 2001).

While the above technique is suitable for configuring a relatively small number of cells, it may be impractical for extremely large matrices, say, ones containing a mole (6.02×10^{23}) of cells. For such large systems, configuration of cells one at a time is impractical. Even at a configuration time of one picosecond per cell, configuring 6.02×10^{23} cells would require over 19,000 years.

If, instead, one configures many regions in parallel, configuration times may be made to vary as the square root (for a 2-D matrix) or cube root (for a 3-D matrix) of the number of cells. In the above example, rather than requiring 19,000 years, the resulting configuration times would be 0.78 seconds in 2-D and 84 microseconds in 3-D.

There is no mechanism inherent in a Cell Matrix for configuring cells in parallel. It is possible, however, to create structures within the Cell Matrix that will perform such parallel configurations. And, since we are implementing a large network of supercells anyway, we can augment our supercell design to perform these parallel configurations. Thus, as the matrix is being tiled with supercells, those that have already been created will participate in the creation of other supercells.

PRE-CONFIGURATION OF CELL MATRIX AS A SUBSTRATE OF SUPERCELLS

This *tiling* phase occurs prior to configuring a target circuit on the Cell Matrix. During the tiling phase, the atomic units of the Cell Matrix (cells) are effectively replaced by the larger atomic units of the final circuit (supercells). This can be viewed as introducing layers of organization on the system, as shown in Figure 2.6. At the conclusion of the tiling phase, the Cell Matrix will be tiled with a collection of supercells. Note that in some sense, from a biological perspective, the role of a Cell Matrix cell corresponds to that of a molecule; the role of a supercell corresponds to that of a biological cell; and the target circuit corresponds to a biological organism.

The tiling process is driven by the transmission of cell configuration strings to a set of cells along an edge of the matrix. As part of this process, it is necessary for supercells to perform two dual functions. When a supercell is surrounded by other supercells, it must simply pass configuration information to nearby supercells. If, however, a supercell is located along the perimeter of a region, i.e., if it is adjacent to unconfigured cells, then it must **use** the configuration information it has received to configure a new supercell. Thus, in some cases, a supercell treats configuration information as data, while in other cases it treats the same information as code. This is the self-duality of a supercell, and is illustrated in Figure 2.7. Figure 2.8 shows the effect of the tiling process, and illustrates why order(n^2) supercells are configured after n steps.

All of the routing of configuration strings to achieve parallel configuration is handled by the supercell network itself. The entire system is still just sent a single copy of the configuration string. This may result in the configuration of a single new supercell, or of 1,000 new supercells in parallel. The input to the system is the same, just a single configuration string. Parallel utilization of this string is achieved by the supercell network itself.

Note that the process of configuring supercells is, in fact, simply the process of configuring individual Cell Matrix cells in an appropriate way so as to create supercells. This is accomplished by having a target cell's D input set to the appropriate value (1 or 0) prior to each tick of a system clock, while the cell is in C-mode (meaning its C input is 1). The generation of the appropriate D input pattern can be achieved in many ways, but is often the result of reading another cell's current configuration, i.e., reading bit patterns from a cell library. Since the reading of the source cell and the writing of the destination cell both occur under the same system-wide clock, there is no need for further synchronization between source and destination. When the source cell has been fully read, the destination cell will also have been fully written. For further details of the low-level configuration of cells, see (Macias 2001).

TESTING OF THE CELL MATRIX

In addition to configuring supercells in parallel, another goal of the tiling phase is fault detection (Durbeck 2002). We introduce the requirement that we want the system to contain only defect-free supercells. Therefore, we will thoroughly test an area of the Cell Matrix for defects immediately prior to configuring it to act as a new supercell. Moreover, this testing is performed by the already-existing supercells that, by the above requirement, can be assumed to be working perfectly. By inducting from a properly-functioning initial supercell, we see that, barring the introduction of run-time faults, the entire supercell network will be defect-free.

This is another form of self-duality: regions of cells are first tested by nearby supercells,

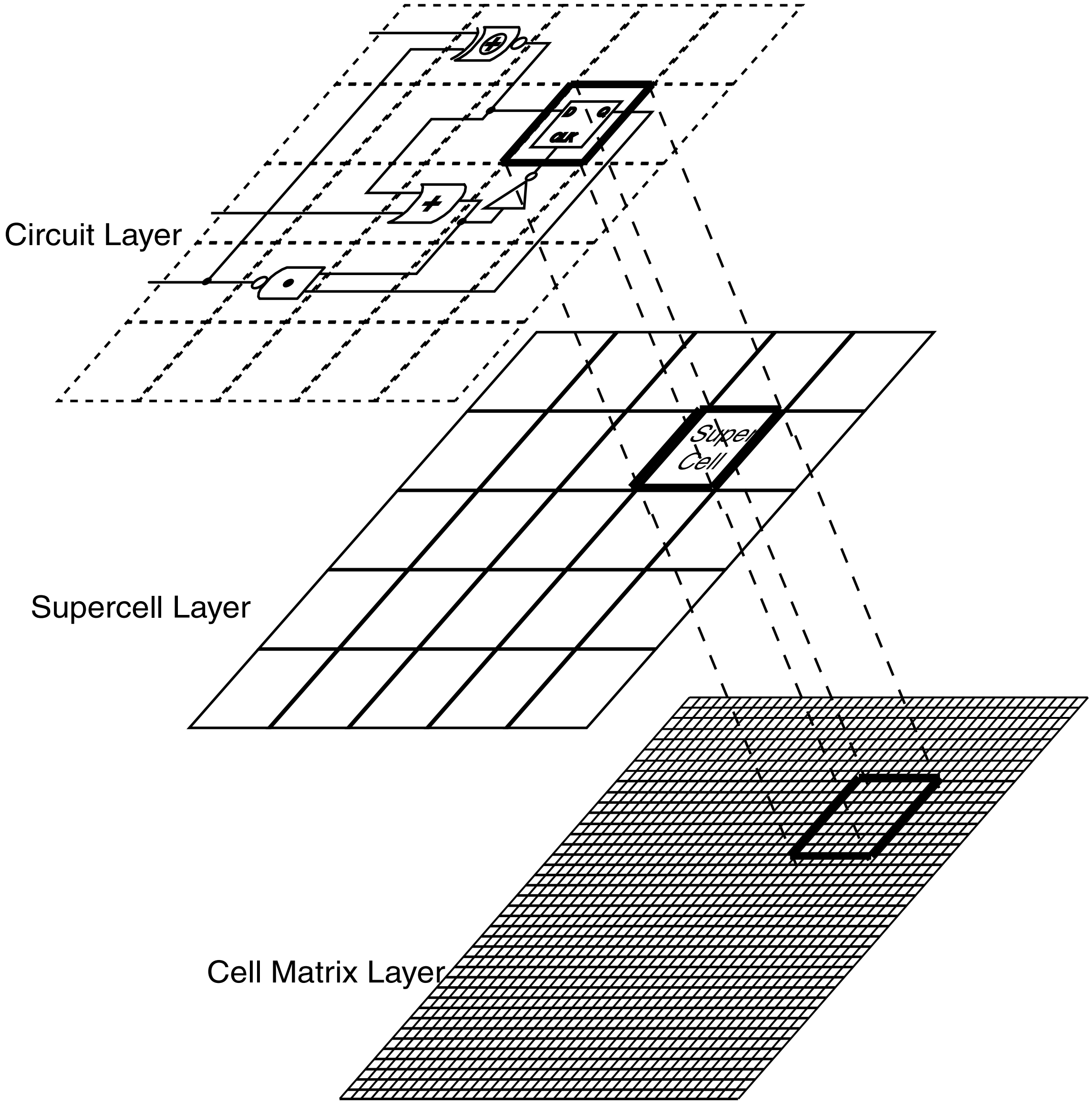


Figure 2.6
Tiling of matrix with Supercells
The tiling can be viewed as a separate Supercell layer implemented above the Cell Matrix layer. Above the Cell Matrix Layer is the Circuit Layer, on which the actual target circuit is implemented. This is purely an organizational view. The Circuit and Supercell Layers are not actually separate from the Cell Matrix Layer.

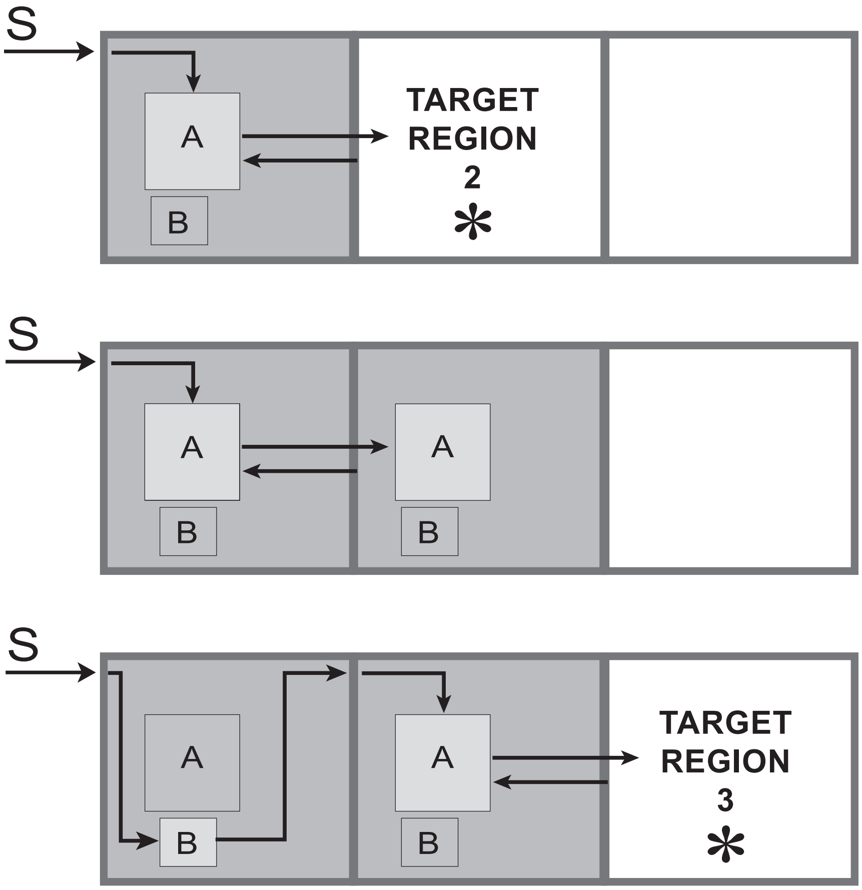


Figure 2.7

Self-Duality in the Configuration versus Use of Supercells

This time sequence from top to bottom illustrates how each Supercell goes from being a target of test-and-build configuration sequences to being the configurer, the source of test-and-build configuration sequences. The configuration string S is repeatedly sent to a region containing three Supercell-sized blocks. In the first time step, the leftmost block has already been configured as a Supercell and is using structures and functions denoted by block A to test the Target Region for defects. In the second time step, the Supercell is configuring Region 2 to be an identical copy of its self. In the third time step, the Supercell has completed its configuration of Region 2 and is now using structures and functions denoted by block B to pass S to Supercell 2. Supercell 2 repeats the process with Region 3.

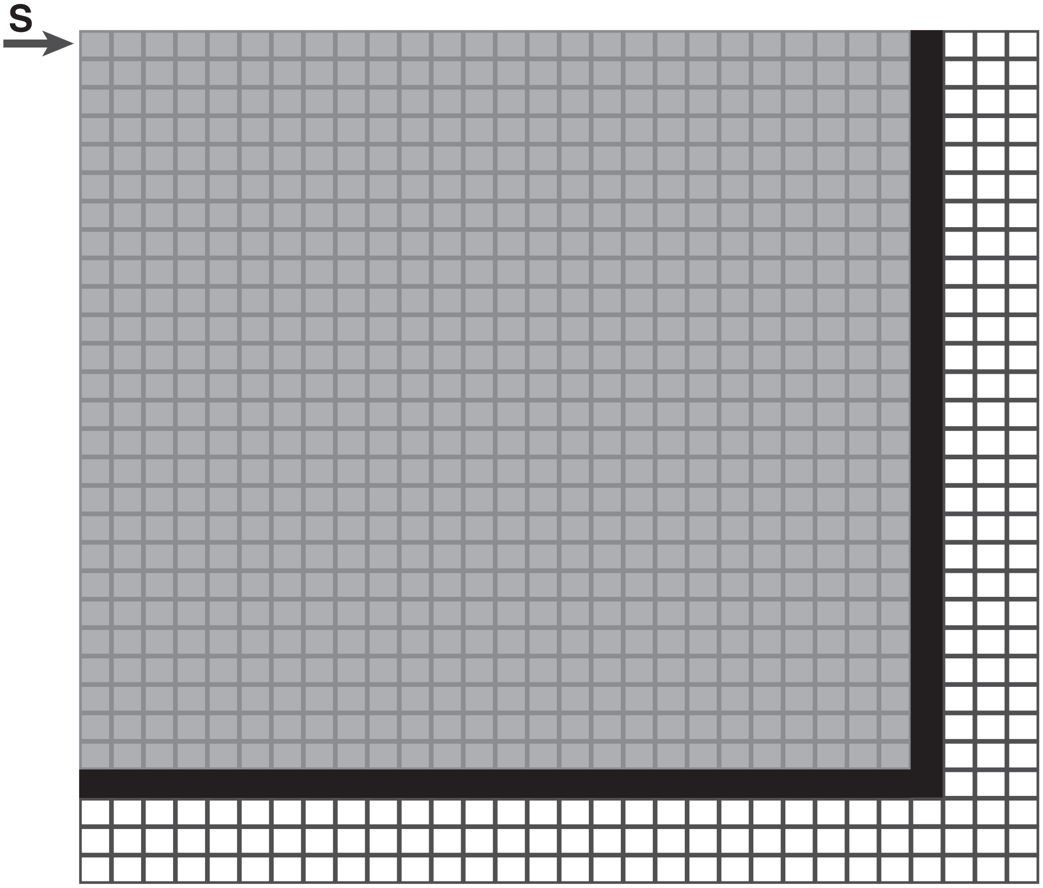


Figure 2.8.a
Parallel Supercell Layout Pattern

The wavefront pattern of increase in the number of regions tested and configured at each repetition of the Supercell definition string is illustrated here. The small squares are supercell-sized regions. Grey supercells have already been configured and are now being used to pass the configuration information to the edge of the configured region. The black region indicates all cells that are tested and configured in the current repetition, where one cycle includes sending signals four times, in the direction of each Supercell's Northern, Eastern, Southern, and Western regions. Thus, all cells in black are configured with 4 repetitions of the Supercell definition. The configuration string is repeatedly pumped into the upper left supercell, as indicated by 'S.' With this starting point, and a lack of defective regions, only the test and build signals to the South and East result in new Supercells.

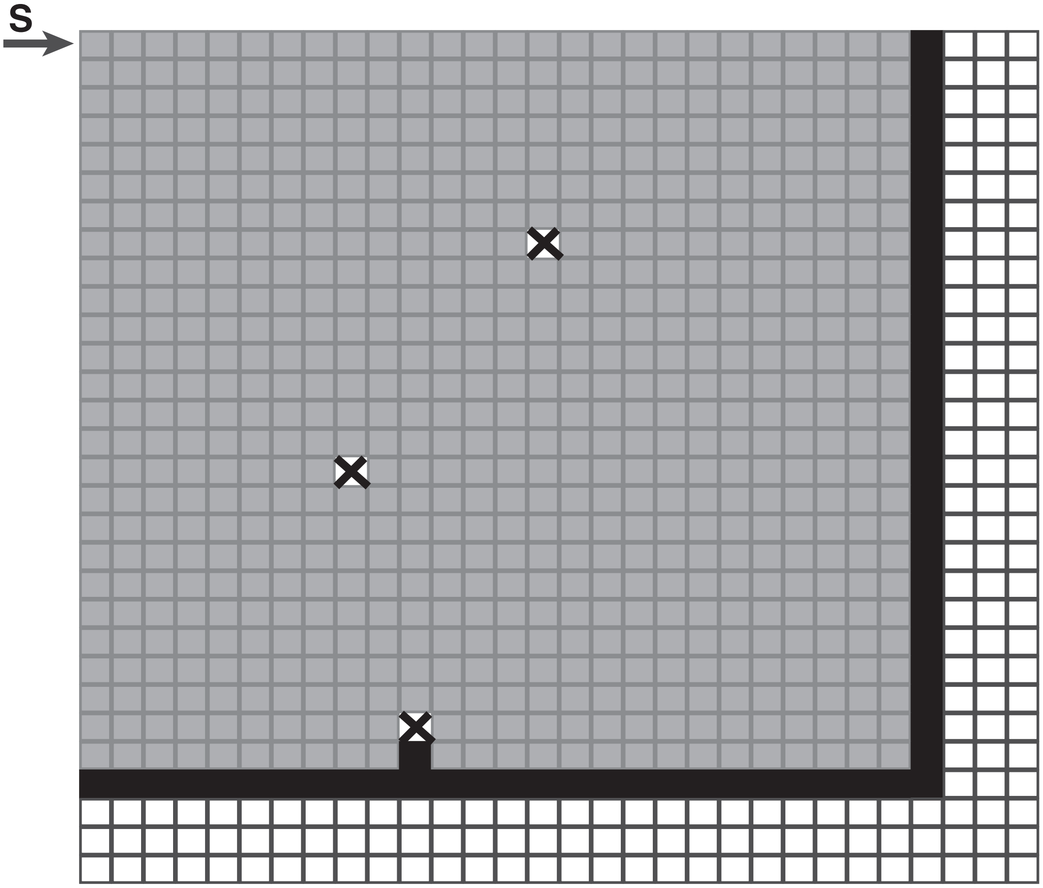


Figure 2.8.b
Parallel Supercell Layout Pattern, Defects Encountered

Grey squares are nondefective regions that have already been configured as Supercells. Black squares are Supercell-sized regions that are tested and configured in this cycle. White squares with 'X' are regions in which a defect was detected. Defective regions cannot be used to configure new neighboring regions. Thus they slow down configuration immediately around them, but not for long, as illustrated by the bottommost 'X.' Note that during this cycle, test-and-build commands are executed to the East first, then to the South. Therefore during the pattern sent to the East, the cell beneath the lowest 'X' is configured by its western neighbor. This new Supercell immediately sends the configuration pattern to the cell below it, when the test-and-build to the South begins. Thus two regions are configured in this column during this iteration, and the smooth edge of the parallel configuration front is restored within this cycle.

and then, having passed those tests, they themselves become supercells, and participate in the testing of other nearby regions, as illustrated in Figure 2.9.

The choice of which tests are applied to cells during the tiling/testing phase is largely independent of the rest of this work, and in practice would be based on separate analysis of the particular hardware implementation in terms of expected failure modes and characteristics. For the present work, we created simple test patterns as a proof of concept. More sophisticated test can be performed in a similar fashion. For example, in one test, the target cell is configured as an inverter. Following this configuration string, a test string is sent to the system, containing a pattern of 1s and 0s. A comparison string is also sent to the system, containing the same pattern but with each bit inverted. The supercell network routes the test string to the region under test (or regions, if multiple regions have been configured and are being tested in parallel). Adjacent to each region under test will be a supercell that reads the return pattern from the region under test, and compares it to the expected return, a substring that is embedded into the configuration string and available to all supercells when testing neighboring regions. If a supercell detects a discrepancy, it will activate a circuit inside **itself** called a Guard Wall (Figure 2.18). The guard wall is simply a collection of cells inside the defect-free supercell, configured in a particular way so as to prevent cells on the defective side from affecting cells on the defect-free side. Thus, guard walls are not specialized hardware within the Cell Matrix, but rather are composed of ordinary Cell Matrix cells. Since the guard wall is contained inside a supercell, and supercells are only built on regions that are defect-free, we can assume that guard walls are also defect free, at least at the time they are configured.

Generally, the test patterns include not only input stimuli and expected output responses, but also configuration commands for building test circuits within the region under test (RUT). Typically, cells at the edge of the RUT are tested first, and are then used to build pathways deeper into the RUT. As more cells are tested, more complex structures can be built to further explore cells within the RUT, testing them in various configurations, from different sides, and so on, as illustrated in Figure 2.10. The result is an extremely thorough testing of the RUT prior to its configuration as a new supercell. Note also that, since this testing is performed by supercells near the RUT, and is performed in multiple regions in parallel. This preserves the speedup which was obtained via parallel configuration during the tiling phase.

PASSAGE TO CIRCUIT LAYER

The primary goal of this research is to implement a desired target circuit on top of a Cell Matrix (Macias 2002). But why should we use supercells to implement this circuit, instead of directly composing the circuit from individual Cell Matrix cells? We have already seen one reason, which is simplicity of the design process. However, the primary reason is **adaptability**. When a circuit is composed directly from cells, the designer must specify exactly which cells will perform which functions in the final target circuit. The exact connection pathways among those cells must also be specified. Thus, the final circuit is specified explicitly. Any implementation of the target circuit using the

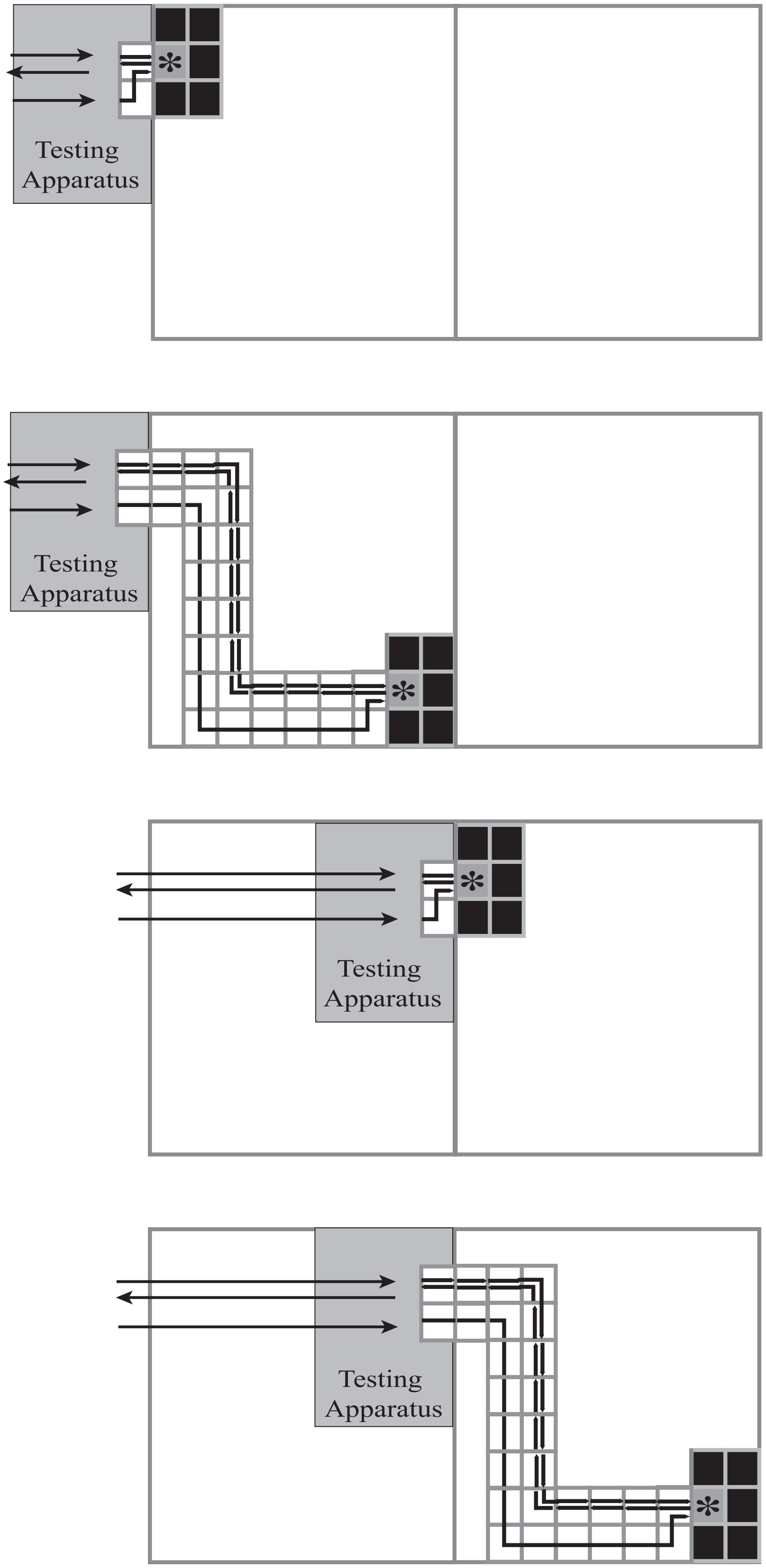


Figure 2.9
Self-Duality in the Testing of Supercell Regions
Sequence of events from top to bottom. The region on the left is being tested; once it has passed the test, it is configured such that it can perform the same test on the region to its right.

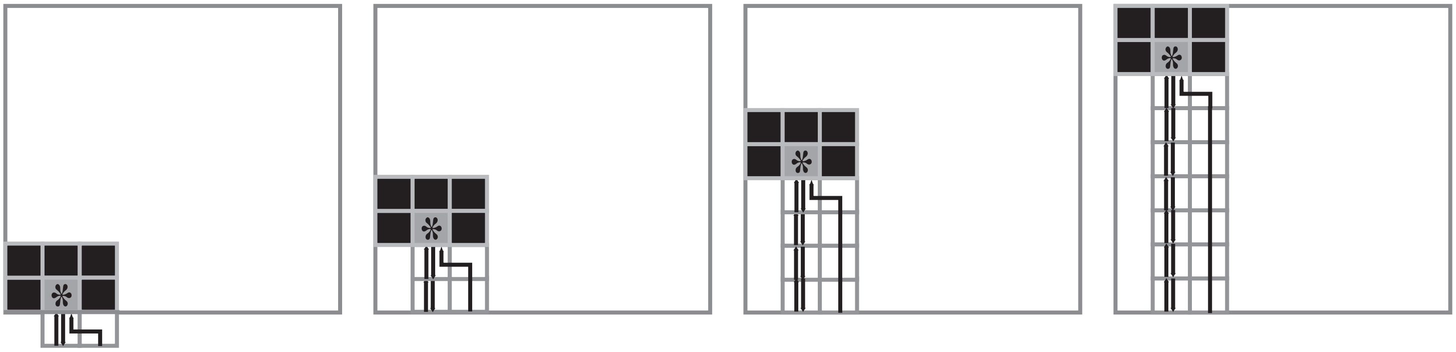


Figure 2.10
Testing each cell within a Supercell for Defects

Testing proceeds through the Supercell region by building wires to each cell and testing all sides directly reachable from the cell. Four stages of the test are shown, with the large white box representing the region that is being tested before being configured as a Supercell. Testing begins with access to a single cell within the Supercell, depicted in the leftmost frame as a grey box with a star symbol. The center grey cell is tested, followed by tests conducted on the black cells. Upon successful completion of the testing, several of the tested, defect-free cells can be used as portions of wires. The wire is extended to them, and the testing starts again at the next cell in the column. Each row of the column is tested fully; however, for briefness only four stages are shown. The black lines are cells that have been configured as pieces of a 2-channel wire.

specification will be identical to any other implementation from the same specification. Such a rigidly-specified circuit will tend to be extremely fragile. If the matrix contains any defective cells, and those cells have been slated to be part of the final circuit, then the final circuit cannot be implemented, because the implementation **must** use those cells, as indicated by the circuit specification.

A more robust approach to circuit implementation is to create a self-organizing circuit that first decides how to implement the target circuit, and then proceeds to actually implement it. When examining this approach in terms of evolutionary selection, one can view the target circuits as being functionally equivalent, but having different forms. The function is rigidly specified by the system itself, by the process through which it self assembles. The form, however, is decided by environmental factors, in particular the state of the underlying hardware in terms of defects and damage. Thus, the environment effectively selects the forms in which the final (fixed) functions are implemented. The atomic unit of this self-organizing system will be supercells.

In order for supercells to function collectively as a self-organizing system, they will need a number of capabilities beyond those of simple cells. Like cells, supercells must communicate with one another. However, whereas cells exchange a single piece of information with their immediate neighbors, supercells must exchange a variety of information, not only with immediate neighbors, but with remote supercells as well. Therefore, supercells requires powerful communication channels among themselves.

Because supercells are to be able to create new supercells, they must contain extensive circuitry for modifying collections of cells, not only adjacent to themselves, but non-adjacent as well. With this power, however, comes a risk. Because supercells can affect large numbers of cells in the matrix, it is possible that a malfunctioning supercell could do considerable damage to other cells in the matrix. Therefore, in the event that a supercell contains cells that are not functioning properly, there must be some way to isolate the effects of that faulty supercell, **without** relying on cooperation from the faulty supercell itself. Thus, supercells must be able to protect themselves from malfunctioning supercells. The detection of faulty regions of the Cell Matrix, and the activation of circuitry inside the supercells to protect them from faulty regions, are part of the preliminary tiling phase of the system (along with the synthesis of the supercells themselves).

The primary function of a supercell network is to self-organize into a target circuit, and this function accounts for most of a supercell's complexity. In order to accomplish this, a supercell needs to perform a number of functions. First, it must be able to decide what part of the final circuit it itself will implement, and then it must differentiate into that circuit element. Second, it must locate other elements with which it needs to communicate. Third, it must establish communication with those other elements. And fourth, it must participate cooperatively with the other supercells in the system, so that each supercell can perform its own required functions. Execution of these steps is referred to as the *differentiation and self-wiring phase* of the system.

DIFFERENTIATION OF FINAL CIRCUIT'S ELEMENTS

All of these steps are governed by a single set of instructions called a genome. For any given target circuit, there is a single genome, regardless of how that circuit will eventually be implemented on the Cell Matrix. The genome describes only the functional behavior of the target circuit. It does **not** specify how that functional behavior should be implemented. The collection of supercells will itself analyze the Cell Matrix hardware, and then determine how to implement the indicated functional behavior. Figure 2.11 illustrates a sample implementation of a target circuit.

The genome can be viewed as an abstract circuit specification. It indicates what elements the final circuit will contain, and how those elements should be interconnected. The supercells will use this abstract circuit specification to perform the differentiation and routing operations described above. Figure 2.12 shows a genome for an example circuit.

It should be noted that each supercell must be identical to every other supercell in the system. This is important, since it allows parallel configuration of supercells (which is essential for efficient operation of the system), and because it allows interchangeability among the system's atomic units (which is essential for maximal fault tolerance). Therefore, each supercell contains the same genome as every other supercell. In order for supercells to differentiate, they must therefore dynamically determine among themselves some unique characteristic associated with each of them. This can be done by noting their position in the matrix, relative to other supercells. Thus, supercells have been given the ability to generate unique IDs. Figure 2.13 shows an initial set of self-assigned IDs in a supercell tiling. These are called Static IDs. They are very easy to assign, as they are based on a supercell's row and column in the tiling. By consulting its neighbors, a supercell can easily assign itself an ID by adding or subtracting from its neighbors' rows and columns. However, because these IDs are position dependent, they cannot be used in the genome, since the genome can contain only position-independent information. Therefore, these Static IDs are used to generate a second set of IDs, called Dynamic IDs. Dynamic IDs are assigned sequentially to supercells in ascending order of their Static IDs. These assignments thus vary depending on the placement of supercells within the matrix, i.e., on the location of defective regions in the matrix. However, regardless of where faults have occurred, it is guaranteed that all consecutive Dynamic IDs will be assigned, up to the number of supercells in the tiling. Figure 2.14 shows the Dynamic ID assignment corresponding to the Static IDs from Figure 2.13.

The genome for the final circuit uses Dynamic IDs to refer to particular circuit elements. Therefore, in order to differentiate, each supercell consults the genome (recall, each supercell has a copy of the complete genome), identifies the section that refers to its own Dynamic ID, and thereby learns what function it is to implement in the final circuit. The supercell then configures a set of cells inside itself to implement that function.

SELF-WIRING OF THE FINAL TARGET CIRCUIT

Self-wiring of the final circuit requires the discovery of pathways throughout the

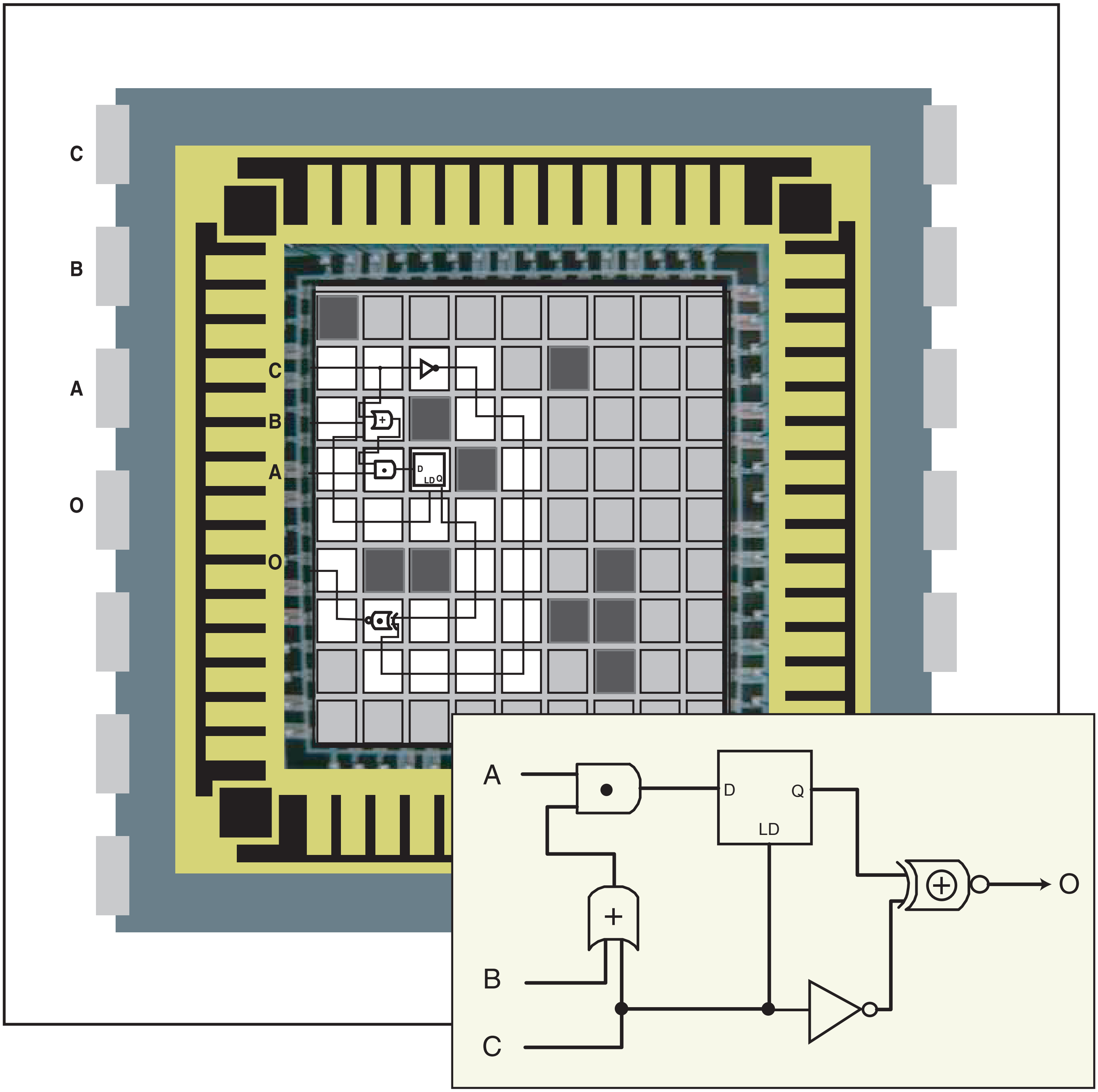


Figure 2.11

Illustration of Silicon Chip with Supercells and Target Circuit

This is intended as an illustration of the technique as it could be used on a custom ASIC silicon chip to detect defects and come up with a layout of the desired circuit despite defective regions within the hardware. The white inset shows a schematic diagram of a simple target circuit with inputs A,B,C and output O. The more detailed illustration depicts a silicon chip and its packaging. The chip contains an array of Cell Matrix cells. The method has already been run on the chip, leaving an array of Supercells, as indicated by light grey boxes, and isolated defective regions, as indicated by dark grey boxes. The functions performed by the Supercells are shown in a schematic representation on the white boxes. A,B,and C can then be passed into the chip from input pins, and O can be read from an output pin.

1	7	0	18	1	1
2	11	0	18	1	1
3	15	0	3	0	0
5	17	1	2	19	9
7	5	2	2	19	9
9	7	1	2	19	9
11	9	2	2	19	9
13	11	1	2	19	9
15	13	2	2	19	9
17	7	15	2	20	1
Node IDs			Function		

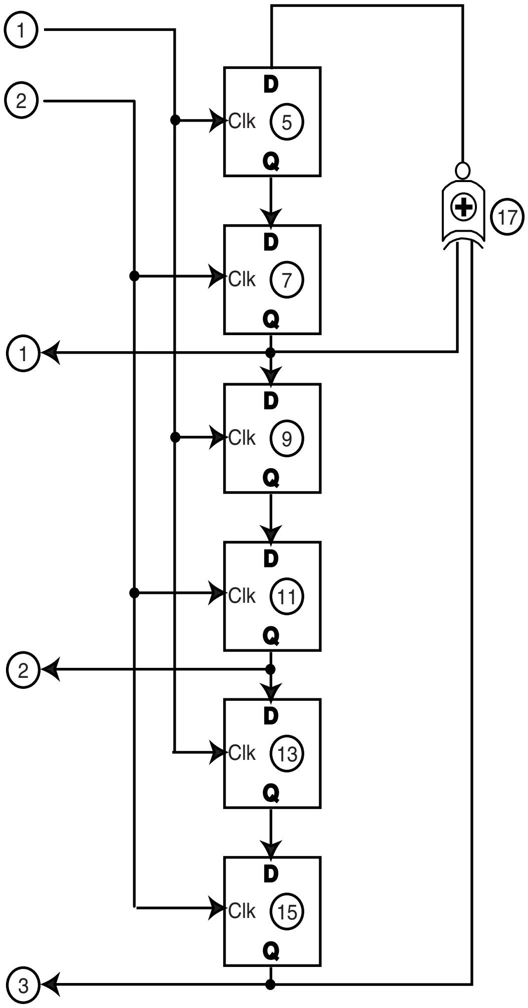


Figure 2.12
Genome for a Sample Circuit
The sample circuit is a linear feedback shift register, which generates pseudo-random numbers. The genome on the left is a string of integers arranged in groups of 6, arranged as a table here where each row is a group. In each group, the first integer N is the node number of the node being described. The second and third numbers are the node numbers of any nodes that send data to node N. The last three numbers indicate the function of node N. For example, the fourth line indicates that node 5 receives inputs from nodes 17 and nodes 1, and that node 5 performs the function [2,19,9] which means a D flip flop. Note that I/O nodes are bidirectional, thus nodes 1 and 2 appear as both input and output nodes.

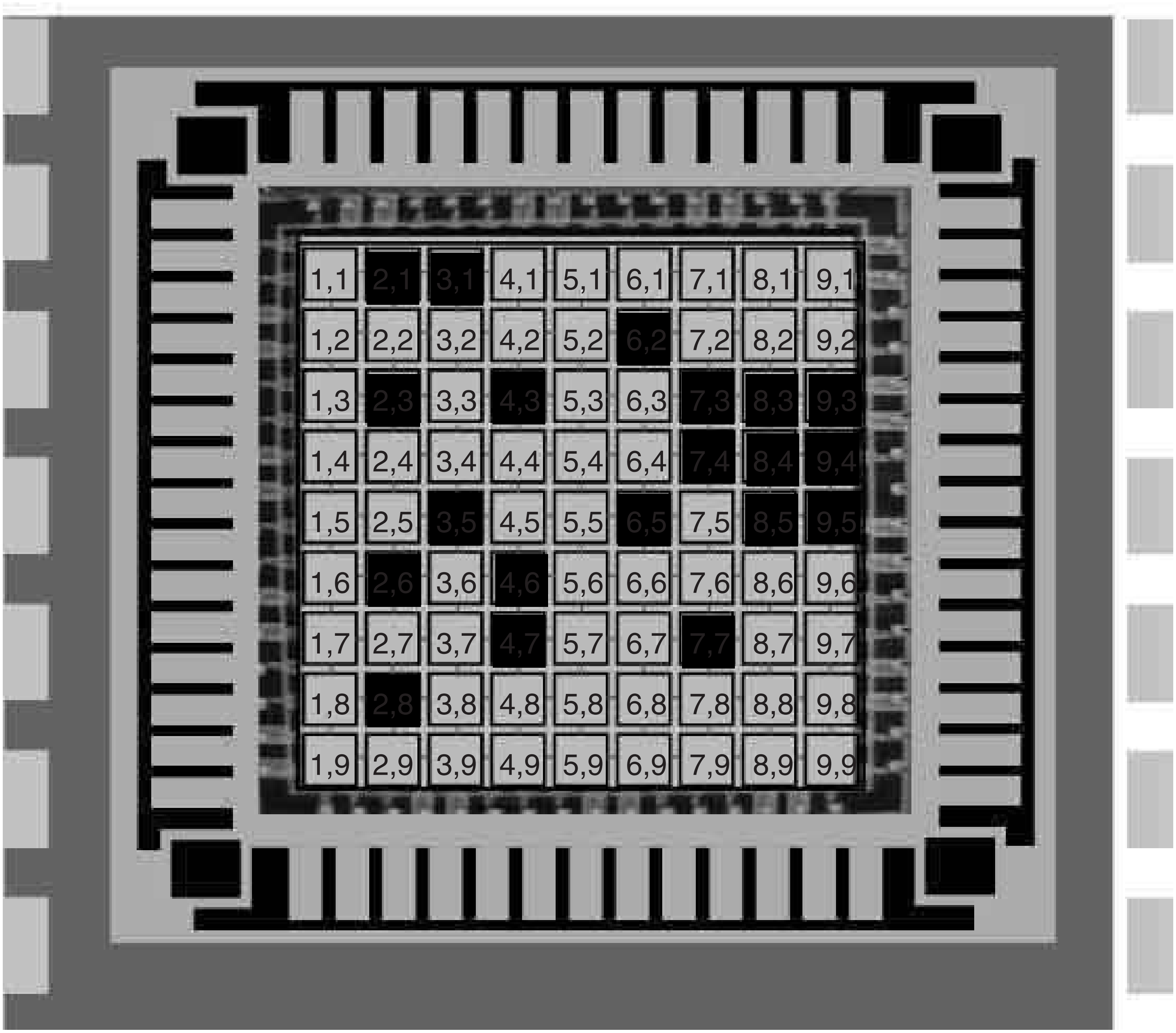


Figure 2.13
Static ID Assignment in a Supercell Tiling
Each grey box is a Supercell. Black boxes indicate faulty regions.
A Supercell's Static ID is based on its absolute position in the tiling,
and thus can not be used in the circuit's genome.
For example, Static ID "3,5" is not present in this tiling.

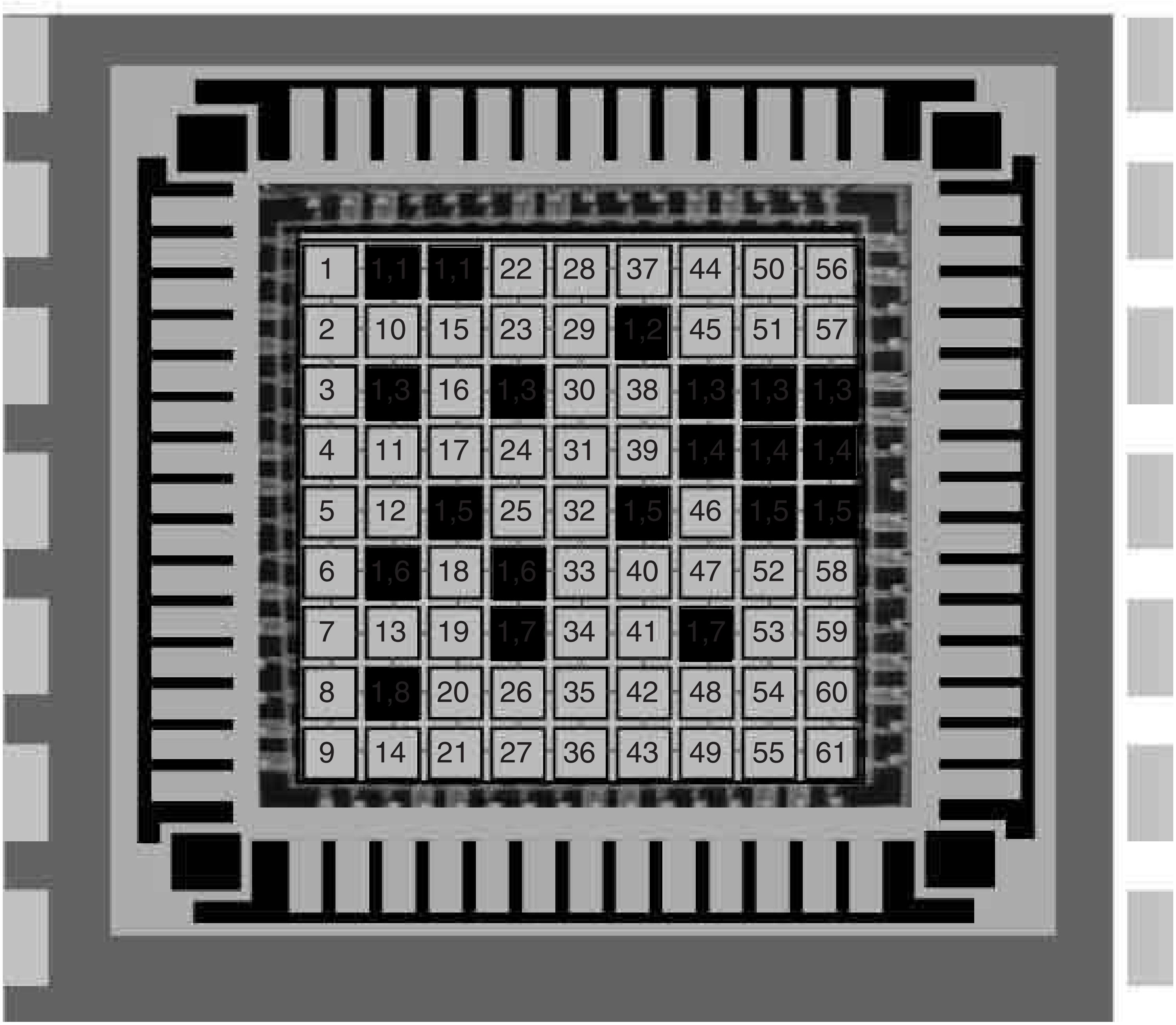


Figure 2.14

Dynamic ID Assignment in a Supercell Tiling
Each grey box is a Supercell. Black boxes indicate faulty regions.
Dynamic IDs are sequential integers, assigned sequentially to Supercells
in increasing order of their Static IDs. Dynamic IDs correspond to node numbers
in the target circuit's genome.

supercell network from one supercell to any other supercell with which communication is required. We modeled our approach on the “ping” command available on most Unix machines. The ping may travel along multiple routes to the destination, but upon reaching the destination, it is returned to the originator with routing information appended, so that the source learns of a path to the destination. This is also similar to the work of (Moreno 2001), that allows already-built pathways to be shared by multiple sources and destinations, whereas the present work creates separate pathways for every source/destination pair.

To wire themselves together and thereby complete implementation of the target circuit, the supercells work sequentially, in the order given by their Dynamic IDs. The first supercell builds a broadcast network, allowing it to transmit information to all other supercells in the system. It then consults the genome, and determines the Dynamic ID of any supercell with which it needs to communicate. A “ping” is then transmitted to all supercells, indicating the Dynamic ID that is being sought. Figure 2.15 shows a sample transmission of this ping. It should be noted that the broadcast network must be loop-free; there should only be one path to any given supercell. This is necessary so that, when the broadcast is terminated, it is not preserved in any feedback loops. The procedure for building the broadcast network ensures that there are no loops in the network.

When the ping reaches the desired supercell, that supercell will respond with an acknowledgment. The acknowledgment is sent to the original supercell along the same route on which the ping was received. However, as the acknowledgment is sent, the supercells that pass it along note the route it is taking, and add that routing information to the front of the acknowledgment. Therefore, when the acknowledgment reaches the original supercell, it will contain routing information that indicates how to reach the original destination of the ping. In Figure 2.16, the echo of the ping is sent from the supercell marked “+” back to the originating supercell, marked “*.” The first supercell to receive this echo (the supercell below “+”) notes its reception from the North, and appends the tag “N” to the beginning of the echo. The next supercell in the transmission chain (the the left of the previous supercell) notes the echo's arrival from the East, and prepends “E” to the echo. This continues, until the echo returns to supercell “*.” At this point, the full echo will contain the string “E,E,N,N,N,N,E,N.” These are precisely the directions which must be taken from “*” in order to reach “+.” In this way, the originating supercell knows how to reach the desired supercell.

Finally, the originating supercell uses this routing information to send a series of channel-building commands through the supercell network. These commands will be passed by some of the supercells as data, but will eventually reach a supercell that interprets the commands as code, and uses them to actually change the configuration of cells within itself. By carefully controlling the configuration of these *steering* cells within intervening supercells, a communication channels can be built between the source and the target of the ping.

In this way, each supercell in turn establishes the necessary communication channels between itself and other supercells, such that in the end, the differentiated elements of the

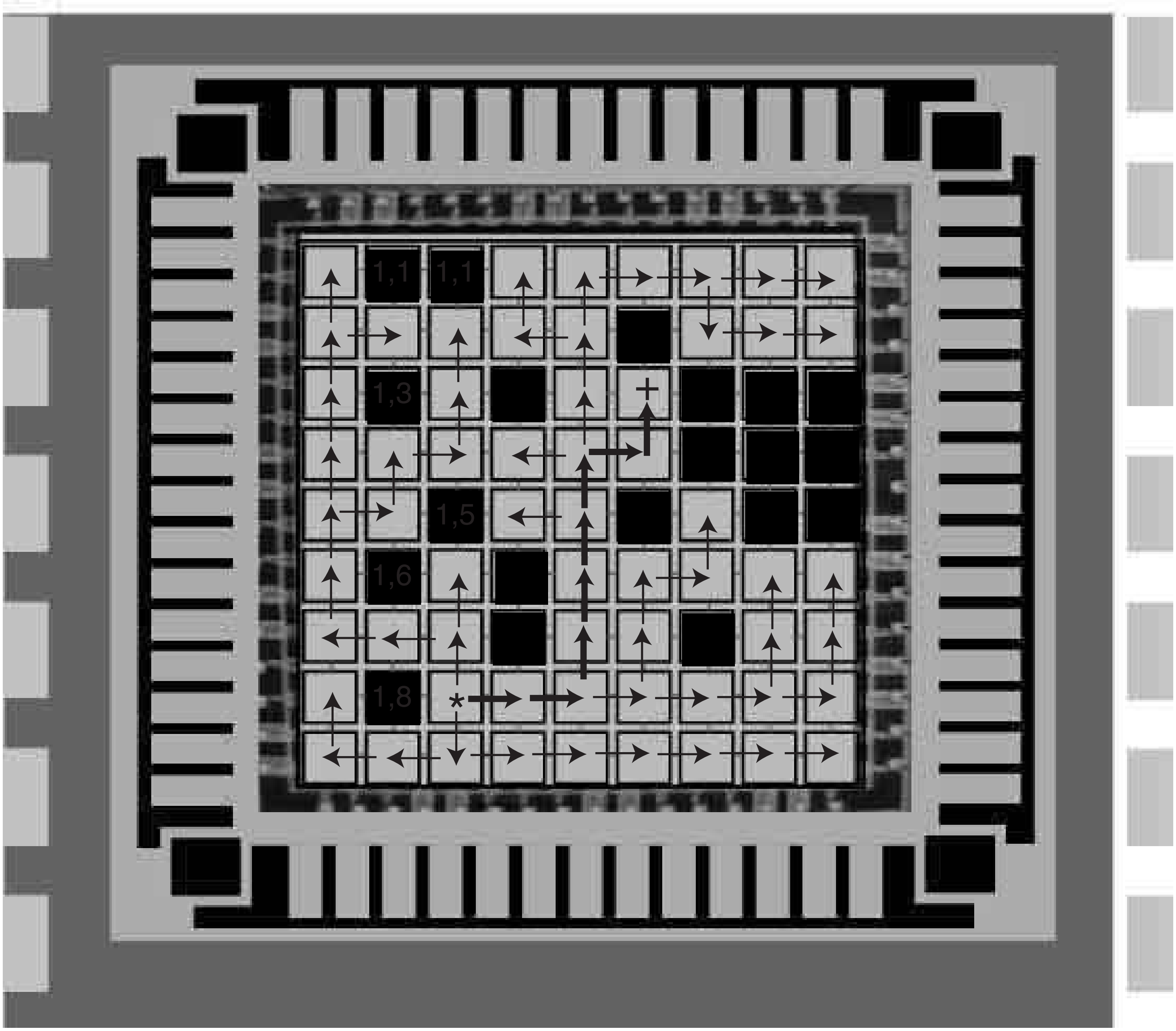


Figure 2.15
Pinging of One Supercell from Another

The source Supercell is marked with an "*" and is looking for
the Supercell marked with a "+"
Arrows show the loop-free broadcast network which has been built by "*"
and allows "*" to broadcast to every Supercell in the system.
The path from "*" to "+" is shown by the heavier arrows.

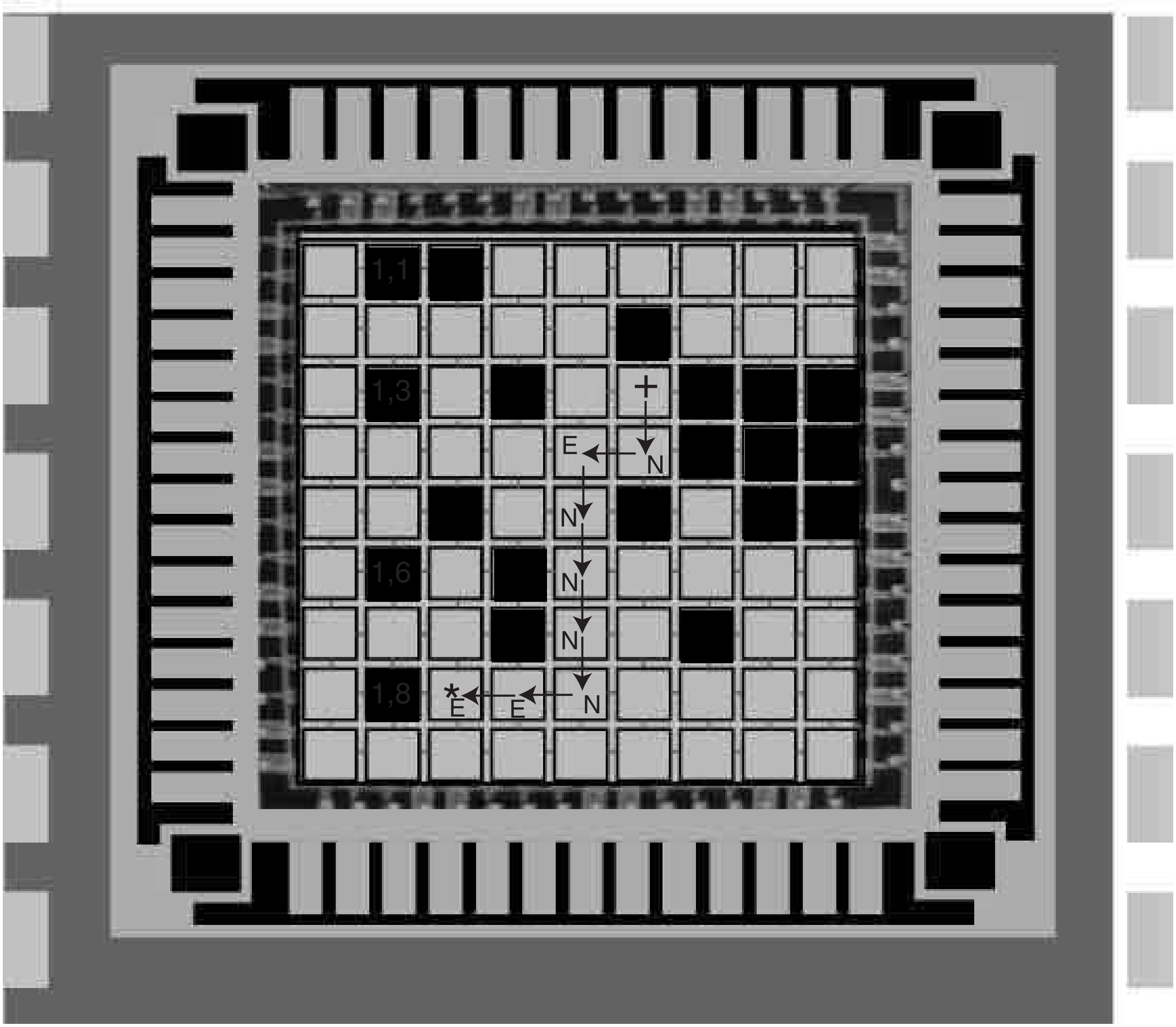


Figure 2.16
Echo of Ping from Target Supercell "+" to Source Supercell "*"

The ping is returned along the same path as the original ping, but in the reverse direction. Each Supercell that helps return the echo appends directional information (indicated in each Supercell) to the beginning of the return packet, indicating from where the echo came into the Supercell. Here, the final echo contains the string E,E,N,N,N,N,E,N.

final circuit are interconnected, as specified by the genome, to implement the final circuit's desired functional behavior.

3. EXPERIMENTS AND RESULTS

To test and further develop these concepts, a series of experiments was performed. First, a supercell meeting the above criteria was designed. Figure 2.17 shows a functional view of this design. The supercell contains a number of subsystems:

- a Genome, which describes the target circuit;
- an ID Arbitrator and Node ID, which determine and store the dynamic ID of the supercell, and thus help determine the role the supercell will play in the final circuit;
- Test Logic, for analyzing Cell Matrix cells, determining when a cell is faulty, and setting Failure Flip Flops to activate Guard Walls, thus isolating the supercell from nearby faulty regions;
- numerous I/O lines for, among other things, configuring nearby regions to be new supercells;
- Trace Generation logic and Dynamic Routing control, to locate those supercells it should connect to and determine pathways to them;
- a Route Synthesizer and Wire Building Cell Library, for configuring cells in the Steering Build Areas of other supercells, thus steering the movement of both configuration commands and pure data through supercells' Pass-through Channels, thereby creating communication channels to remote supercells (this is another example of self-duality in a supercell);
- a Differentiator and Functional Block Library for customizing an internal Functional Block that will represent an element of the final target circuit; and
- various Clocking and Control logic for directing all of these operations autonomously.

Next, a Cell Matrix layout was generated to implement the above circuit. This was done using the Cell Matrix Graphical Layout Editor (Cell Matrix Corporation 2001). The Cell Matrix implementation of the above supercell is a 270x270 circuit, containing 72,900 individual Cell Matrix cells. The design was then compiled into a binary representation, and the resulting compiled circuit description was used to generate a set of configuration strings that, when sent to an empty Cell Matrix, cause the above tiling, differentiation and wiring phases to occur.

These configuration strings were then sent to a simulated 2000x2000 Cell Matrix, and their effect on the Matrix was observed. Finally, the outcome of the process, the target circuit operating on top of the supercells, was tested by setting inputs to different values, and observing the resulting outputs. For the circuits on which we tested this system, the final circuit always worked perfectly.

FAULT TESTS

Next, we added simulated faults to the Cell Matrix simulator, including stuck-at-0 faults, stuck-at-1 faults, and faults where cells cannot change mode. We then requested that certain cells inside the matrix be simulated as being faulty, and sent into the matrix the same configuration strings from the earlier tests. In these cases, we now observed that supercells located near the faulty cell(s) detected the presence of faults, and activated their guard walls to isolate the effects of those faults. Figure 2.18 shows how supercells work in tandem to isolate a faulty region of the Cell Matrix. Again, following the tiling phase, the system's supercells differentiated and self-wired, and again implemented perfectly working versions of the target circuit.

To further test the ability of the system to detect failures in the Cell Matrix itself, we also performed tests on physical (as opposed to simulated) hardware. Using a different type of reconfigurable hardware, a Xilinx ® Spartan-II^{TM} FPGA (Xilinx Corporation 2001) controlled by a PC using a BurchED prototyping board (Burch 1997), we were able to implement a small Cell Matrix in hardware. The implementation was done using Xilinx's Webpack, which is a freely available design and place-and-route package.

Instead of creating a perfect layout, we modified the implementation so that a single cell had one of its inputs stuck-at-0. We then sent the supercell configuration strings into this hardware Cell Matrix, and observed that the cell testing section indicated a failure in the Cell Matrix, at precisely the cell that contained the “fault.” Figure 2.19 shows the schematic for the Cell Matrix that was implemented on top of the FPGA, including the input which is stuck-at-0.

It should be noted that while the Cell Matrix *architecture* is fault isolating, this particular hardware implementation of a Cell Matrix on top of an FPGA is **not** actually fault isolating, because the underlying hardware (the FPGA), like most hardware, is extremely fault sensitive. Should even a single defect arise in the FPGA itself, the Cell Matrix will likely not longer function properly, because the FPGA will no longer be able to implement the architecture correctly. To realize the fault isolation potential of the Cell Matrix architecture, it needs to be implemented directly in silicon, or on some other medium that is not inherently fault sensitive.

In addition to the above tests, we wanted to more thoroughly test (on the simulated Cell Matrix) the ability of the system to self-organize into the final target circuit despite the presence of faults. We decided to perform these tests separately from the tiling-phase tests. This decision was made because:

1. the tiling phase uses a high degree of parallelism, and therefore runs very slowly on a sequential simulation (e.g., on a PC);
2. the outcome of the tiling phase is relatively predictable, as it depends only on the location of the faults and the location of the edge cells to which the configuration strings are applied; and
3. the tiling phase is logically separate from the differentiation and self-wiring phase, and thus it is safe to evaluate each separately.

In view of this, for our subsequent tests, we switched to simply tiling the matrix using the simulator's ability to directly manipulate any cell within the system. Following this tiling, we then let the system's supercell's perform their differentiation and self-wiring phases.

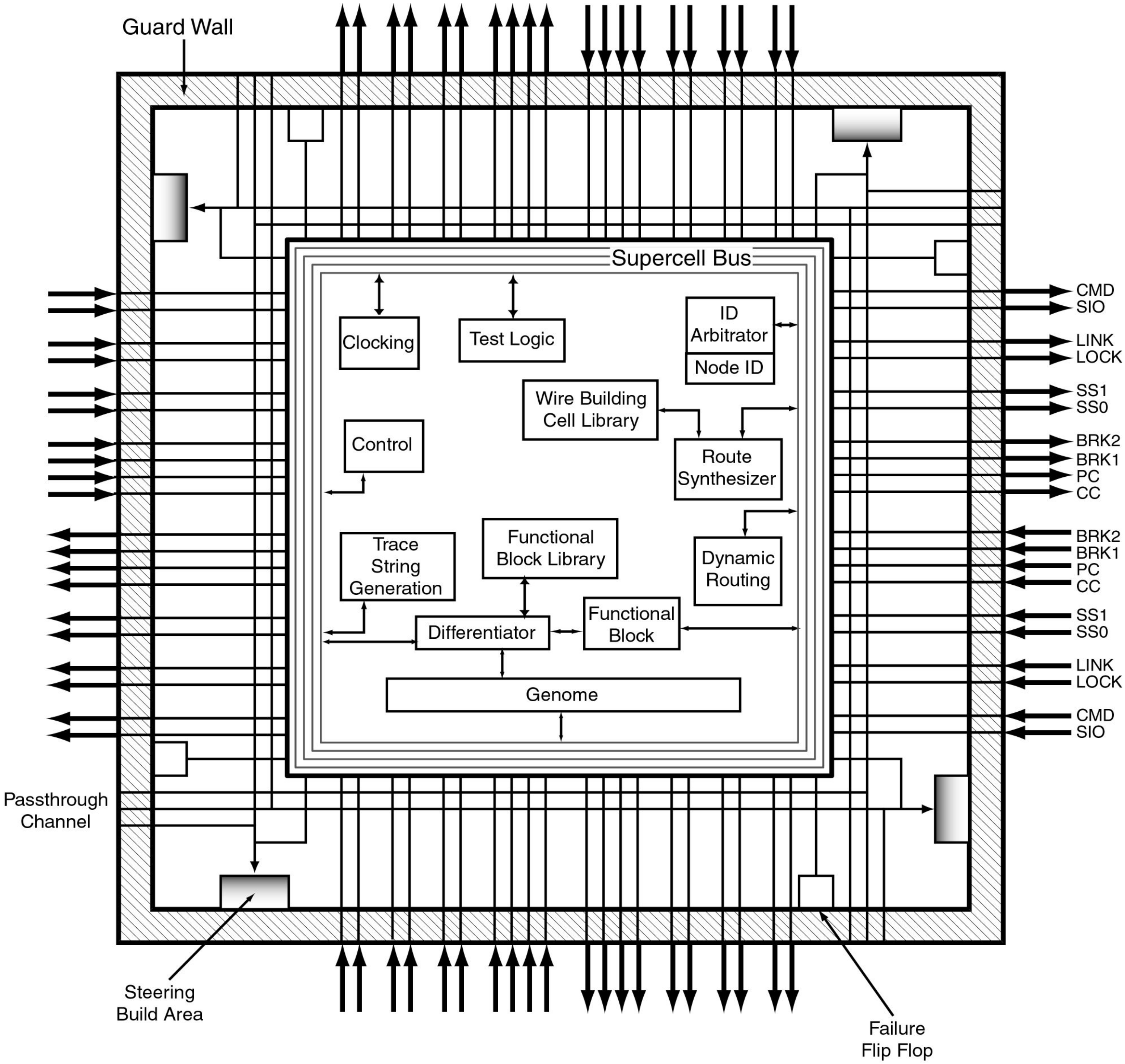


Figure 2.17
Supercell Block Diagram

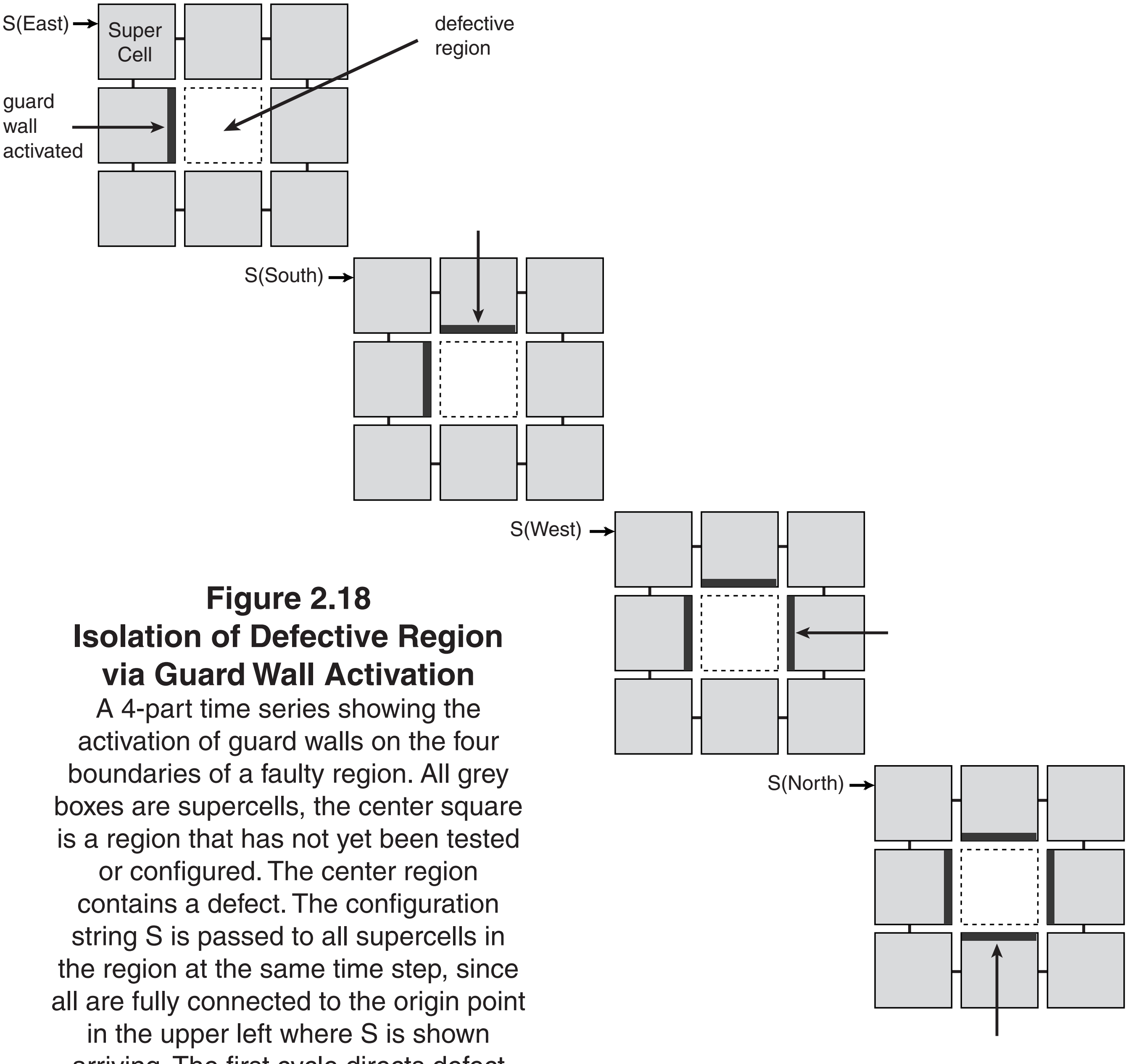


Figure 2.18
Isolation of Defective Region via Guard Wall Activation

A 4-part time series showing the activation of guard walls on the four boundaries of a faulty region. All grey boxes are supercells, the center square is a region that has not yet been tested or configured. The center region contains a defect. The configuration string S is passed to all supercells in the region at the same time step, since all are fully connected to the origin point in the upper left where S is shown arriving. The first cycle directs defect testing to the East, which causes the first cell, to the West of the defective region, to detect it and raise its guard wall. The second cycle directs fault direction to the South, which causes the cell to the North of the defective region to detect it and raise its guard wall. After test-and-build commands to the West and North, the faulty region is completely isolated, no signals from it can escape and disturb the behavior of the neighboring good hardware.

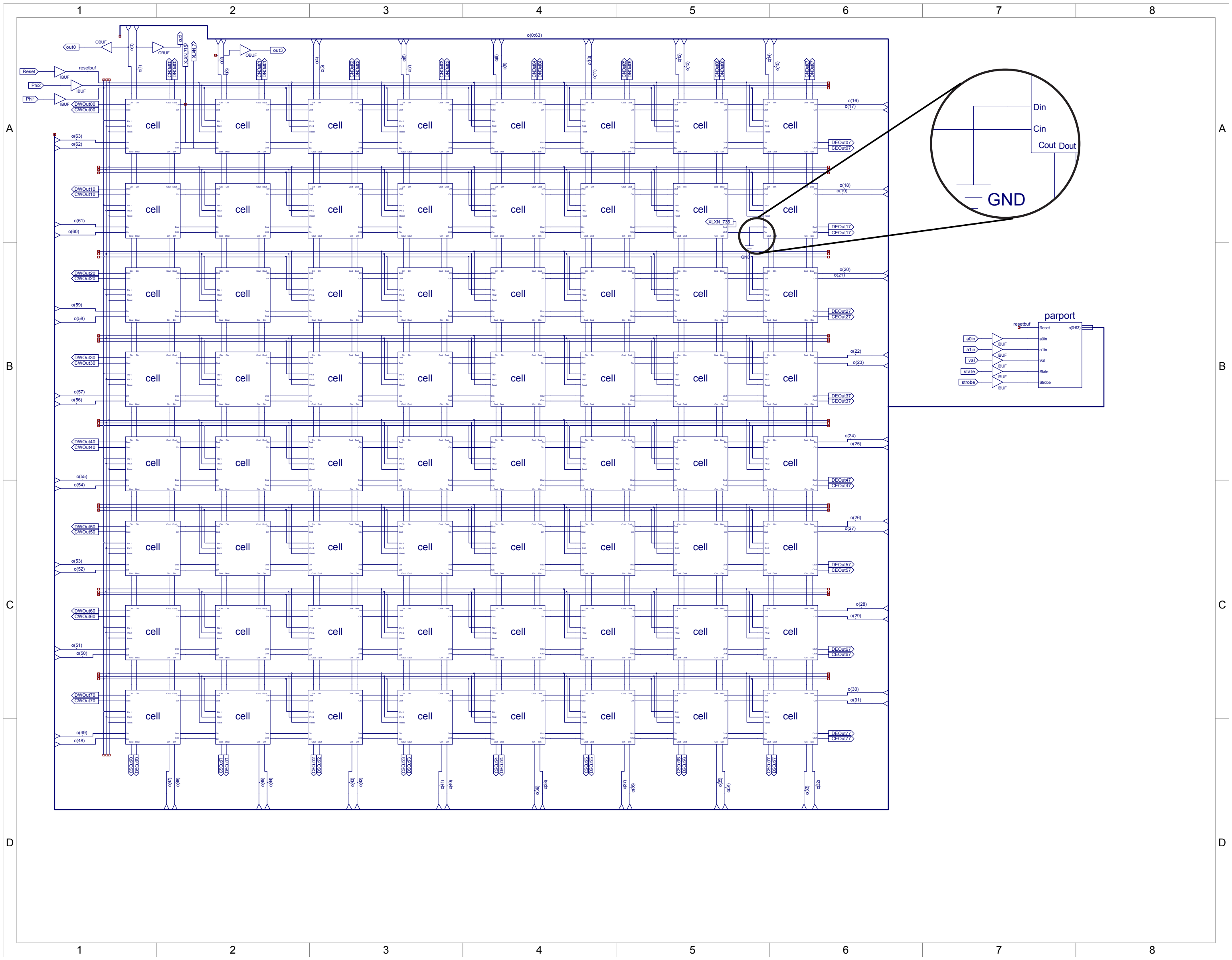


Figure 2.19

Cell Matrix Implementation on Top of an FPGA

64 cells are arranged in an 8x8 pattern.
Cell in row 2, column 8 has its D input shorted to ground
to simulate a defect in the Cell Matrix's hardware. An enlargement
of the defect is shown in the upper right (circle).

We further enhanced the simulator to allow the recording of certain state information from each supercell, including information from the Functional Block, Failure Flip Flops and Steering Build Areas. We then created post-run analysis tools to read this state information and produce a graphical display of the system's final state. This allowed us to analyze how the system worked around faulty areas and wired together the final circuit. We also automated a process for setting the final circuit's inputs to different combinations, and recording the resulting behavior of the circuit.

TEST RESULTS

Figure 2.20 shows a sample of the system's output following the differentiation and self-wiring phase of the system. In the test illustrated by this figure, a network of 25 supercells was used, with five of them being defective, as shown. The lines and circles show the mapping of connections from one supercell to another. When a line enters a supercell, it travels around a circle to emerge on another side. The color coding is simply to aid in following a signal's path around a circle.

For example, in Figure 2.20, the supercell in row 4, column 1 ([4,1]) sends its output to the right, to supercell [4,2]. [4,2] sends this data to its right, to [4,3] (follow the purple circle), and [4,3] is the final recipient. Similarly, supercell [1,4] sends it output to the right to [1,5] (purple circle), then to [2,5] (red circle), to [3,5] (red circle), to [3,4] (blue circle), to [3,3] (blue circle), to [2,3] (turquoise circle), to [2,2] (blue circle), to [1,2] (turquoise circle), to [1,1] (blue circle), and finally to [2,1], where the path terminates.

Using this test setup, we analyzed the behavior of the system in its synthesis of a one-bit adder, a three-bit counter, and a three-bit Linear Feedback Shift Register (pseudo-random number generator), under a variety of fault conditions. Figure 2.21 shows the results of one set of tests, where an increasingly difficult pattern of faults was presented to the system. For readability, the output graphs such as that shown in Figure 2.20 have been simplified, with all pathways from one side of a supercell to another being shown as a single line. In all cases, the system was able to autonomously self-organize into a working final circuit, despite the presence of faults in the Cell Matrix.

4. DISCUSSION

The techniques described appear to provide a Cell Matrix hardware/software system the ability to reorganize its own digital circuitry to avoid its own defective hardware. This makes it possible for a system to modify itself in response to damage, relocating critical functions to safer locations and regrowing signal pathways. The capability to exhibit these types of responses to events that harm it is unusual for synthetic systems. This work is significant in that it invests the synthetic system itself with these capabilities, giving it higher degrees of the autonomy and robustness that living systems exhibit. Historically, digital electronics do not self-repair; recovery from damage is extremely rare, and it is

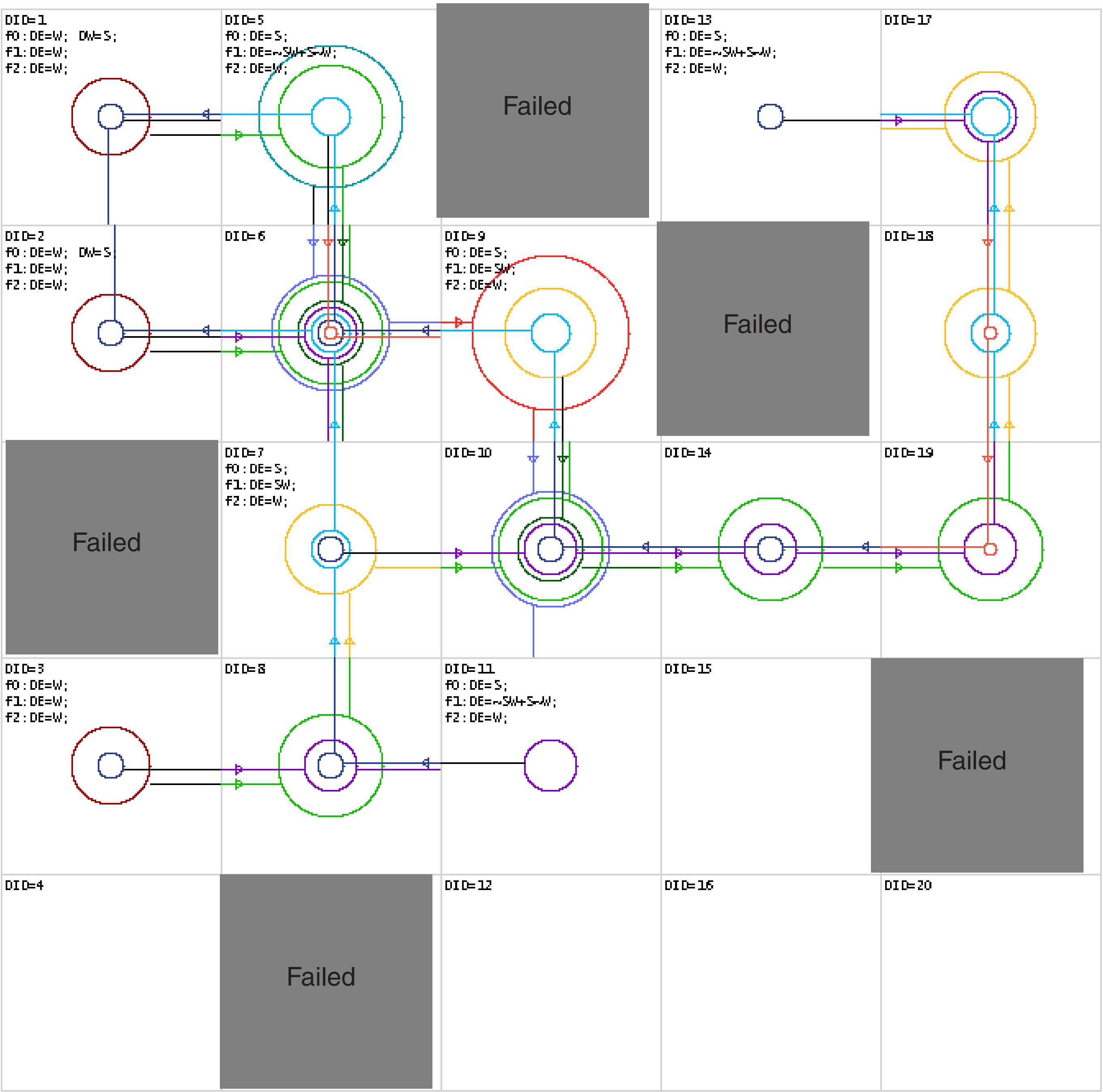


Figure 2.20

Sample circuit routing in the presence of faults

Each block is a Supercell, or a region containing one or more defects.
Lines indicate a communication pathway from one Supercell to another.
Lines that terminate at the edge of a Supercell are inputs to that Supercell's functional block.
Colored circles show the passage of a signal from one side of a Supercell to another.
Lines emerging from a circle with no input to the circle come from the Supercell's functional block's output.

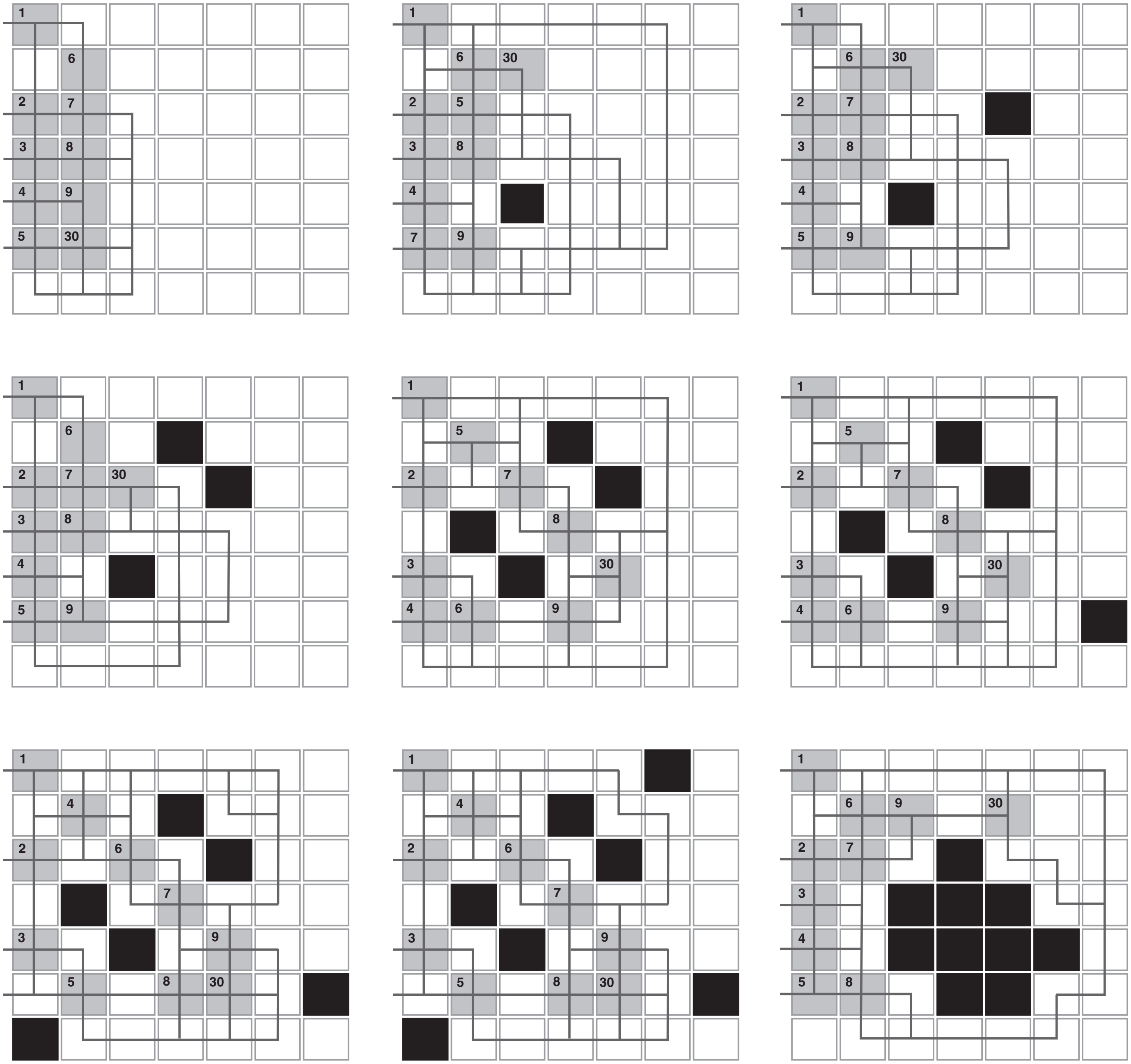


Figure 2.21
Results of High Level Simulations
The high level algorithm simulation was run for the same circuit genome with different numbers and locations of defects. Each 7x7 array of Supercells represents one run. Black regions contain defects and were not configured as supercells by the algorithm. Grey and white regions are Supercells, grey indicates those that were differentiated into functional logic blocks. Numerals indicate which logic block. Lines represent one or more wires the algorithm creates between components. The ability to detect and work around defective areas is evident, as is the variability the defects the faults introduce.

generally achieved through a high degree of external intervention: a human or other complex external monitoring system detects failures, diagnoses the problem, attempts to repair and reinitialize the system, and if unable, replaces the damaged components. Viewed as a whole, this traditional system of hardware combined with a repair mechanism is quite complex and labor-intensive.

If instead all of a system's digital electronics could self-repair, it would be possible to limit the work done to maintain the system. The system could handle most of it itself. External intervention would be required just to monitor subsystems for proper or improper functioning, suspend or notify those processes that are dependent upon a failed subsystem, direct a failed subsystem to repair itself, and resume normal functioning once the subsystem has completed the repair. In the event that the failure is too massive and the subsystem cannot recover, conventional techniques for replacing components could be used (not all of which require a repair person). If a higher degree of autonomy were required for the system, it could be invested with the ability to send the configuration string that builds the missing subsystem to a piece of spare unused Cell Matrix hardware kept in reserve. In either case, it seems reasonable to expect that having subsystems repair themselves frees up the larger system's time and resources, because it would be able to focus most of its efforts on just those subsystems that have failed in a massive way and need to be replaced.

This delegation of the details of self-repair to each subsystem would seem to permit the development of much more complex systems than are possible today – ones comprised of many more subsystems. Each subsystem can be designed in the same manner, as a system with many self-repairing subsystems, as appropriate for the complexity of the subsystem and the physical and computing resources it possesses. Figuring out exactly how to achieve this image is no small engineering feat. What we have provided is simply an existence proof for adaptive, self-repairing Cell Matrix systems, and one technique for achieving such systems.

Other techniques are conceivable, and might be more appropriate than the one described here, depending on the specifics of the situations and constraints it addresses. A significant aspect of the supercell technique is that it renders defect-tolerant a large part of the defect testing and accommodation process. More specifically, checking for faults is done by supercells, which are created only on hardware that has successfully tested as defect-free. There is no external testing system that also must be checked and maintained defect-free. Within the Cell Matrix hardware, there are no critical sections. Any region can become a supercell, and any supercell can test its neighbors for defective hardware. The process of testing all the hardware is performed by way of a completely distributed web of supercells. Thus, defects encountered early in the process do not prevent the rest of the regions around them from being tested, because the signals are automatically routed around them. The process of wiring up circuit components is also done by this highly connected web of surpercells, providing many options for routing each signal, and thus increasing the likelihood that routes will be found. The only piece that must be carefully maintained outside the Cell Matrix is the configuration string that contains the definition of the supercell and the North-South-East-West building pattern. This can be compactly represented and can be stored on a nonvolatile piece of hardware that is made robust using standard redundancy or other conventional techniques for remote systems, or transmitted/communicated to the system when needed.

To use this technique in a real-world setting, it would need a way to detect failures in the implemented circuit in order to know when to rebuild. Standard voting schemes, built-in test, or other strategies can be employed for this. The determination that a new fault has occurred somewhere in the system is outside the scope of this work, and is left to separate techniques. Once a failure has been detected, the Cell Matrix needs to be re-initialized. The Cell Matrix architecture specifies a single system-wide RESET signal that returns all cells' internal programs to a default power-up state. Once the matrix has been re-initialized, the configuration strings are sent into the matrix, causing the tiling/testing phase to execute. The end of the configuration strings contain a GO signal, which terminates the tiling/testing phase, and causes operation of the system to switch to the differentiation/self-wiring phase. At the conclusion of that phase, the system generates its own COMPLETE signal, indicating to external systems that the synthesized target circuit has been successfully implemented, and is ready for use.

DISCUSSION OF EXPERIMENTS AND RESULTS

The experiments conducted provide a reasonable amount of evidence that our approach works. We would have a more compelling case if we showed a defective chip on which, using this technique, we were able to lay out a target circuit that functions properly, and we intend to attempt this. However, the way the problem was formulated, there is a simple electronic target circuit that is the end result of performing the supercell technique on Cell Matrix hardware. That circuit is expected to function perfectly, and this expectation can be tested. When it is tested, it is tested not in a simplified model, but on top of a detailed, low-level simulation of Cell Matrix hardware. Thus, proper function of the target circuit implies proper function of the layers below it: the low-level Cell Matrix hardware that comprises each piece of each supercell, as well as the supercell structure's ability to implement the circuit elements and wires. Further, the target circuit was created in the final stages of a simulation of the entire supercell process. Thus, proper function of the final target circuit implies that the process by which it was obtained also functions properly.

We used only a small number of test patterns in testing for faults, but the fault testing technique is entirely general. The process of building pathways throughout a region under test (RUT), so as to access each cell inside the RUT from all sides, is independent of the choice of tests. Similarly, the method of testing regions in parallel, and then using defect-free regions to test subsequent regions, is also independent of the particular tests applied at each point.

As to what sorts of tests can be applied, there is no restriction on the test patterns that can be used. For example, a cell can be configured to output whatever bit it receives. By sending all 0s, one can ensure that all memory locations are capable of storing 0. Likewise for a pattern of all 1s. A pattern of 10101010....1010 ensures that adjacent bits are not shorted together. If the cell's internal program is physically stored in a 16x8 arrangement, a test pattern of 111111110000000011111111....00000000 would ensure that no bits are shorted across adjacent rows. If an analysis of the physics of the fabrication process suggested that a certain anomalous event would occur when a particular bit pattern was stored in or received by a cell's circuitry, that pattern could be sent to the cell being tested, and the resulting behavior observed by the supercell performing the test.

The test circuits we implemented inside the supercell were extremely simple, and basically combinatorial in nature; that is, the circuit cannot recognize behavior that depends on the history of previous parts of the test. Rather, a test consists simply of an input pattern and an expected output pattern. However, there is no reason a supercell's testing logic can not be made more sophisticated, depending on the nature of the faults that are to be detected.

In the scope of this work, we limited circuit size to fairly small test circuits as target circuits. This was mainly for the sake of speed, as we were simulating the behavior of this (parallel) system with a sequential program running on a single-CPU machine. Therefore, we worked with small tilings of supercells, and limited our test cases to those that would fit on such small tilings. However, the technique appears to scale well, and, had we the benefit of parallel hardware, should be extremely efficient for larger target circuits.

More extensive testing was done with a higher-level simulation of the supercells' self-wiring techniques. These tests allowed larger tilings to be simulated, and in these larger tests, the system continued to perform well, successfully routing circuits in the presence of faults. As expected, the success of the system falls off slowly as the number of faults increases, until a high fault density is reached, at which point the system has great difficulty implementing the target circuit.

It should be noted that the particular supercell that was implemented in this work is only an example of the general supercell technique. Many design choices were made in this implementation: the size of the supercell; the nature of the functional blocks that each supercell implements; the number of independent pathways that can be made through a supercell; the structure of a target circuit's genome; and so on. There are also numerous ways in which the self-routing of the system can be achieved. The implemented supercell employs a greedy algorithm that takes the shortest path between each pair of supercells that need to communicate. Because of this, routing of the entire final circuit could fail, simply because of the order in which supercells elect to self-wire. This can be improved upon greatly, using any of a variety of place and route algorithms. However, for a proof-of-concept, we opted for the most straightforward approach, which was a greedy shortest-path algorithm.

RELEVANCE

The large size of a supercell greatly diminishes the immediate applicability of this work. However, fabrication techniques are undergoing intense research and development, and may facilitate practical use of this work at a later date. As fabrication techniques move closer to the single nanometer range, the scalability of this technique may become critical. Despite the fact that we mostly tested small target circuits, this work method was designed to scale up to large, practical, real-world systems, given a sufficiently dense manufacturing technology for building the underlying Cell Matrix hardware.

The smaller the supercell, the sooner its applicability to manufacturing. The size of a supercell is not an absolute: many inefficiencies exist in the present work that can be improved upon. The 270x270 size of the final supercell is largely a function of the prototypical nature of this research. Much of the supercell implementation consists of unused cells. This is simply a reality of having to create the Cell Matrix implementation by hand, due to the lack of sufficiently advanced compilation tools. Creating a design that was easy to debug and edit was more important than creating one that was as small as possible.

Also, in the current implementation, the functional block that is implemented by a supercell is extremely small, consisting of only three Cell Matrix cells. This means that to create a final circuit with the equivalent of “n” Cell Matrix cells could require 270x270x(n/3) Cell Matrix cells, i.e., the final circuit is roughly 24,000 times larger than a native implementation directly on the Cell Matrix. This extremely inefficient situation can be improved greatly by increasing the size of the functional blocks inside the supercells. If, for example, the functional block was roughly 1,000 cells instead of 3, the size of the supercell might increase from 72,900 cells to 75,000, but implementing a circuit of “n” cells would now require 75,000x(n/1000) Cell Matrix cells, i.e., the final circuit would only be 75 times as large as a native implementation. Functional blocks consisting of thousands of cells might, for example, correspond to simple multi-bit arithmetic units, small memory systems, basic signal processing modules, and so on.

Another issue to consider is how the supercell size impacts the ability of the system to tile the Cell Matrix with supercells. Certainly, if the goal is to implement a circuit that requires only 25 Cell Matrix cells, it makes little sense to begin by searching for 25 270x270 defect-free regions. It would be simpler to simply find a defect-free 5x5 region and implement the target circuit directly on top of that.

The situation is different for large circuits though. Consider a fabrication technique that can manufacture cells with only one fault per million cells. Since a supercell requires 72,900 defect-free cells, you can fit approximately 13 supercells in a one million cell region. Assuming a single defect in that region, 12 of your supercells would be perfect, and one would be defective. This amounts to a 8% defect rate for supercells. Viewed differently, if your *cell* defect rate is one out of a million, then you can configure supercells on 92% of your substrate.

Assuming a defect rate of one per million cells is achievable, you can thus use supercells for implementing arbitrarily large circuits, **much larger than a one-in-a-million rate**

would normally permit. In other words, the supercell approach puts a cap on the size of defect-free regions that are required to be abundantly available. Once the fabricating technology can support large numbers of defect-free regions of the required size, you can immediately scale up to arbitrarily large target circuits. The size of the target circuit does not further impact the manufacturing requirements.

IMPLICATIONS

While Cell Matrices fabricated using today's silicon fabrication technology are not dense enough to make this research practical for real-world applications, the implications of this work used with future manufacturing technologies are manifold. In a realm where the equivalent of transistor switches can be constructed from, say, a few hundred carbon atoms, a mole of carbon could contain $6x10^{21}$ switches, sufficient to implement $6x10^{18}$ cells. Even with the current low-density supercell design, this would allow implementation of $8x10^{13}$ supercells. If each supercell implemented a 3-cell functional block, the resulting system could implement *fault tolerant* circuits roughly 40 million times larger than today's most complex CPUs (Durbeck 2001b, Macias 2002).

In this way, the supercell technique discussed in this paper stands to benefit greatly from the advent of nanotechnology and other atomic-scale fabrication technologies. However, we believe this work can also be used *by* the nanotechnology community to assist in the development of extremely dense fabrication technologies. Because the Cell Matrix is composed of very simple atomic units, arranged in a very regular, infinitely scalable organization, it makes an attractive manufacturing target. And, because fault-aware, fault-tolerant circuits can be implemented on top of a Cell Matrix using techniques such as supercells, it should be possible to implement properly-functioning circuits on imperfect platforms, **and** to use those circuits to test and explore the underlying substrate. Thus, there can be a symbiosis between new manufacturing techniques that allow denser Cell Matrices, and Cell Matrices that assist the debugging of new manufacturing techniques.

Traditional reconfigurable logic suffers from at-best linear configuration times, and therefore would likely require prohibitively long setup times were they to be manufactured at an Avogadro scale. In contrast, because the supercell tiling itself performs the configuration of new supercells, we achieve better-than-linear configuration times. For a 2-D tiling, n^2 supercells can be tiled in roughly n steps. For 3-D, n^3 supercells can be tiled in n steps. This brings system configuration times back into a manageable realm. Likewise, since testing is also performed in parallel, it should be practical to test a huge, Avogadro-scale matrix by performing only 84 million tests.

Moreover, because the configuration strings that are sent to a Cell Matrix depend only on the target circuit, and not on the location of faults in the matrix, a single set of configuration strings could be sent to a number of physically distinct Cell Matrices in parallel. Even though each matrix would receive exactly the same configuration string as every other matrix, each would respond in its own way, depending on the location of faults within its own hardware. In the end, each Cell Matrix would be configured uniquely, to accomodate each Matrix's specific defects, despite each having received the same strings. This too has tremendous implications for the manufacturing process.

FUTURE WORK

The next step in this research in terms of proving this method is to run experiments with silicon chips that contain arrays of Cell Matrix cells. We will experiment first to see whether the technique finds defects that we purposely introduce on the hardware; then, given sufficient information about the hardware manufacturing technique, we will modify the test patterns to capture those defects most likely to occur with the technique, and then run experiments with a large number of silicon chips to determine whether the technique finds the defects that are present, and creates working circuits despite the presence of defects. We are particularly interested in how this could loosen the requirements for highly novel manufacturing techniques such as those being developed at the scale of 1-10 nanometer transistor/feature sizes: extremely advanced silicon manufacture, molecular nanotechnology, quantum dots, and others. If the manufacturing technique need not be perfect, then this may open up the possibilities for choosing a wide variety of manufacturing techniques, because there will probably be more techniques that are *good* than that are *perfect*. Even techniques that are relatively poor may still be useful if they are extremely dense (extremely small electronic components, close packing of cells) or extremely inexpensive, and this is a way to utilize them.

We would also like to work with foundries and other researchers to use some of these techniques to study a variety of manufacturing techniques that are currently in use or in development. Of interest is whether this distributed, local defect detection mechanism helps to improve the manufacturing processes themselves. Our expectation is that it will provide highly localized information that can be used to find most or all the defects present in the hardware, and that it can give more concrete information from which it can be determined what measures to take to improve the manufacturing process. We are interested in working with physicists, chemists, material scientists, and others in trying to work back even further than the functional, software-level structure, seeing what techniques might be helpful if used in conjunction with mechanical or chemical processes to grow the physical structure, such as growth according to some description delivered by the supercells.

Quite a few extensions or modifications of this approach merit investigation. Working with three dimensional layouts of cells appears to be a way to provide many more ways to connect any two components, and thus, may prove to be much more robust to extensive damage. This could be achieved by extending the supercell definition to a three dimensional Matrix and running experiments to compare the differences between 2-D and 3-D systems in terms of types and locations and numbers of defects. If intending to use this method in commercial electronic systems, we must first analyze its efficacy with particular target applications and particular hardware manufacturing techniques: does this technique work well for the kinds of damage encountered in the situation, and the kinds of defects to which the manufacturing technique is prone? Given the dimensions of a supercell, what is the minimum system size at which this cost is superseded by the increase in adaptivity? How might this technique be adapted to the particulars: ideas include making the supercell fit within a larger or smaller area, increasing the size of the functional block implemented by each supercell, varying the placement of supercells instead of the regular tiling used here.

Extending this general method to other types of problems besides circuitry on defective hardware also merits investigation. Ideas include working with different kinds of “payloads” other than a circuit definition in the form of a genome, use of different specialized structures and functions (e.g., what block A in Figure 2.7 does) to see whether we can figure out how to generalize the method or apply it to new problems such as growing circuits in three dimensions, getting supercells to perform local optimizations of a system, making the level at which the target circuit now lies have more capability for self-analysis. In the current work it is simply a static circuit. This is a way to pursue our interest in utilizing local, dynamic features in systems, in response to internal or external events, situations, and phenomena.

Another variant of supercells might be created that does not require an externally-supplied configuration string. Instead, the supercell definition would include how to build a large network of supercells to cover the hardware. This would require that we merge the supercell definition with previous work on self-replicating circuits. This may lead to an extremely robust system: if there is some sort of hardware breakdown and only one supercell survives with its definition intact, it could quickly restart the tiling process to all sides and reestablish the supercell tiling. Possibilities that corruption of the supercell definition would lead to a supercell that could still replicate itself, but that had some sort of mutation, could result in a means by which the supercell's behavior drifts and evolves.

We believe the supercell method is generalizable to at least several other problems that can be similarly formulated.

ACKNOWLEDGEMENTS

Much of this work was funded by NASA SBIR Contract NAS2-01049. The authors gratefully acknowledge the support of Ames Research Center and NASA. We also wish to thank Chrystopher L. Nehaniv, René te Boekhorst, Kerstin Dautenhahn and the attendees of the Evolvability and Individuality Symposium (St. Albans, 2002) for many useful and stimulating discussions. The authors also gratefully acknowledge the many useful comments and suggestions from the reviewers of this manuscript.

REFERENCES

Amerson, R., Carter, R.J., Culbertson, W.B., Kuekes, P. and Snider, P.(1995):Teramac -- Configurable Custom Computing. In *Proceedings of the 1995 IEEE Symposium on*

FPGA's for Custom Computing Machines, pages 32-38.

Bradley, D. W. and Tyrell, A. M. (2000): Immunotrics: Hardware Fault Tolerance Inspired by the Immune System. In Miller, J.F., Thompson, A., Thomson, P. and Fogarty, T. C. (eds): *Evolvable Systems: From Biology to hardware, Third International Conference, ICES 2000*, pages 11-20.

Burch Electronic Design(1997):http://www.burched.biz/index.html

Burks, A. W. (1961): Computation, Behavior and Structure in Fixed and Growing Automata. In BehavioralSc, 6, pages 5-22.

Cell Matrix Corporation(1999):http://www.cellmatrix.com

Cell Matrix Corporation(2000):Downloadable Simulator. At http://www.cellmatrix.com/entryway/products/software/simulator.html

Cell Matrix Corporation (2001):Portable Layout Editor. At http://www.cellmatrix.com/entryway/products/software/layoutEditor.html

Culbertson, W., Amerson, R., Carter, R., Kuekes, P. and Snider, G. (1996): The Teramac Custom Computer: Extending the Limits with Defect Tolerance. In *Proc. IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems*.

Durbeck, L. and Macias, N.(2001a):Self-configurable parallel processing system made from self-dual code/data processing cells utilizing a non-shifting memory. US Patent #6,222,381.

Durbeck, L. and Macias, N.(2001b):The Cell Matrix: an architecture for nanocomputing. *Nanotechnology* **12**:217-230.

Durbeck, L. and Macias, N.(2002):Defect-tolerant, fine-grained parallel testing of a Cell Matrix. In Schewel, J., James-Roxby, P., Schmit, H. and McHenry, J. (eds): *Proc. SPIE ITCom 2002 Series 4867*, pages 71-85.

Heath, J. R., Kuekes, P. J., Snider, G. S. and Williams, R. S. (1998): A Defect Tolerant Computer Architecture: Opportunities for Nanotechnology. In *Science*, V.280, No 5370, pages 1716-1721.

Jackson, A. H. and Tyrell, A. M. (2001): Asynchronous Embryonics. In Keymeulen, D., Lohn, J., Stoica, A. and Zebulum, R. S. (eds): *Proceedings of the 3rd NASA/DoD Workshop on Evolvable hardware*, pages 201-210.

Macias, N.(1999):The PIG Paradigm: The Design and Use of a Massively Parallel Fine Grained Self-Reconfigurable Infinitely Scalable Architecture. In Stoica, A., Keymeulen, D. and Lohn, J. (eds): *Proc. The First NASA/DOD Workshop on Evolvable Hardware*, pages 175-180.

Macias, N.(2001):Circuits and Sequences for Enabling Remote Access to and Control of Non-Adjacent Cells in a Locally Self-Reconfigurable Processing System Composed of Self-Dual Processing Cells. US Patent #6,297,667.

Macias, N. and Durbeck, L.(2002):Self-Assembling Circuits with Autonomous Fault Handling. In Stoica, A., Lohn, J., Katz, R., Keymeulen, D. and Zebulum, R.S. (eds): *Proc. The 2002 NASA/DOD Conference on Evolvable Hardware*, pages 46-55.

Mange, D., Sipper, M., Stauffer, A. and Tempesti, G. (invited paper, 2000a): Towards Self-Repairing and Self-Replicating Hardware: The Embryonics Approach. In *Proceedings of the 2nd NASA/DoD Workshop on Evolvable Hardware,* pages 205-214.

Mange, D., Sipper, M., Stauffer, A. and Tempesti, G. (2000b): Towards Robust Integrated Circuits: The Embryonics Approach. In *Proceedings of the IEEE, Vol. 88:4,* pages 516-541.

NASDA Office of Research and Development(1996):Research on Space Fault Tolerance. In *NASDA Report No. 49.*

Moreno, J. M., Sanchez, E., Cabestany, J. (2001): An In-System Routing Strategy for Evolvable Hardware Programming Platforms. In Keymeulen, D., Lohn, J., Stoica, A. and Zebulum, R. S. (eds): P*roceedings of the 3rd NASA/DoD Workshop on Evolvable hardware*, pages 157-166.

OptiMagic(1997):Programmable Logic Jump Station. At http://www.optimagic.com/

Ortega-Sanchez, C., Mange, D., Smith, S. and Tyrrell, A.(2000): Embryonics: A Bio-Inspired Cellular Architecture with Fault-Tolerant Properties. In *Genetic Programming and Evolvable Machines 1(3),* pages 187-215.

POE(2002): Webpage for the Poetic Project. At http://www.poetictissue.org/

Prodan, L., Tempesti, G., Mange, D. and Stauffer, A.(2001): Embryonics: Artificial Cells Driven by Artificial DNA. In *The 4th International Conference on Evolvable Systems*, pages 100-111.

Stauffer, A., Mange, D., Tempesti, C. and Teuscher, C. (2001): BioWatch: A Giant Electronic Bio-Inspired Watch. In Keymeulen, D., Lohn, J., Stoica, A. and Zebulum, R. S. (eds): *Proceedings of the 3rd NASA/DoD Workshop on Evolvable Hardware,* pages 185-192.

Vanstone, S.A. and van Oorschot, P.C.(1989):*An Introduction to Error Correcting Codes with Applications*. Kluwer Academic Publishers, Boston.

Xilinx Corporation(2001):Xilinx Datasheet, *Spartan-II 2.5V FPGA Family: Functional Description*.

von Neumann, J. (1966): The Theory of Self-Reproducing Automata. Burks, A. W. (ed), University of Illinois Press.