A Greedy Algorithm for Tolerating Defective Crosspoints in NanoPLA Design

Thesis by

Helia Naeimi

In Partial Fulfillment of the Requirements
for the Degree of
Master of Science

California Institute of Technology
Pasadena, California

2005
(Submitted May 11, 2005)

Acknowledgements

I would like to thank all the people who made this work possible by their valuable advice and support. First I like to thank my adviser, Professor André DeHon for his constant help and support in my research and his encouragement for me.

I would also like to thank Amir F. Dana for all the valuable discussions and conversations we had about this research and his constructive feedback. I also received valuable feedback from Michael DeLorimier, Nachiket Kapre and Michael Wrighton, which helped me to improve this presentation.

This research was funded in part by the DARPA Moletronics program under grant ONR N00014-01-0651 and N00014-04-1-0591.

Abstract

Recent developments suggest both plausible fabrication techniques and viable architectures for building sublithographic Programmable Logic Arrays using molecular-scale wires and switches. Designs at this scale will see much higher defect rates than in conventional lithography. However, these defects need not be an impediment to programmable logic design at this scale.

We introduce a strategy for tolerating defective crosspoints in PLA architecture. We develop a linear-time, greedy algorithm for mapping PLA logic around crosspoint defects. The mapping algorithm matches the PLA logic to the defect configuration of each device.

We note that P-term fanin must be bounded to guarantee low overhead mapping and develop analytical guidelines for bounding fanin. We further quantify analytical and empirical mapping overhead rates. Including fanin bounding, our greedy mapping algorithm maps a large set of benchmark designs with 13% average overhead for random junction defect rates as high as 20%.

Contents

Chapter 1

Introduction

Recent work shows how to build nanoscale Programmable Logic Arrays (nanoPLAs) using the bottom-up synthesis techniques being developed by physical chemists [1] [2] [3] [4]. With these bottom-up techniques, it is possible to build features (*e.g.* wires and programmable junctions) without relying on lithography. This provides a path to sublithographic feature sizes (*e.g.* 3nm diameter nanowires have already been demonstrated). As such, these techniques provide a path to continue the advance of field-programmable technology beyond the end of the traditional, lithographic roadmap (*e.g.* [5]). These techniques may also make it possible to achieve small feature field-programmable devices without the full expense of the finest line lithographic processing.

Nonetheless, nanoscale features, both in the sublithographic and lithographic arenas, come with a new set of challenges. Notably, as devices become smaller, they are constructed from fewer and fewer atoms and molecules. Since individual atoms behave statistically, this means we have higher variance in the shape and makeup of our devices, and a higher likelihood that devices are simply unusable. Designs at this scale **must** be defect tolerant. This, and other aspects of sublithographic assembly techniques, suggest that all devices we build at these scales will be reconfigurable.

Hewlett-Packard has recently demonstrated an 8×8 crossbar using molecular switches at the crosspoints [3]. In the HP crossbar, they observed that 85% of the crosspoint junctions were programmable (15% were defective). The HP crossbar is an early laboratory prototype, and we expect these defect rates to decrease. Nonetheless, we are unlikely to achieve 100% crosspoint yield at this scale using these kinds of bottom-up, statistical fabrication techniques. If we have a 100×100 crosspoint array and randomly distributed faults, essentially every row and every column will contain a defective junction. Even at a 95% crosspoint junction yield rate, we will likely find at most one row or column which has no crosspoint defects.

From the above paragraph its clear that we need a mapping algorithm that works properly in the presence of defect junctions. In the following section we introduce some of the previous work that has been done in the area of the defect-tolerant mapping for nanotechnology devices.

1.1 Related Works

Lee *et al.* [6] have demonstrated a technique to program the HP memory array architecture [3], when there is a random distribution of defects over the junctions. The memory architecture consists of $2^n \times 2^n$ array, used as memory and two $2n \times 2n$ arrays, used as a multiplexer and a demultiplexer to read and write to memory. They assume that there are spare nanowires so that the design will be defect tolerant.

In their approach first they find a $2^k \times 2^k$ defect-free array for memory ($k < n$). Next they try to program the $2k \times 2k$ multiplexer and demultiplexer allowing defects in the junctions. In order to program the arrays to multiplexer and demultiplexer, a specific program will be mapped to the arrays. In order to program an array each of the nanowires in the array will be programmed to a specific logic. They try to find a matching from the nanowires logic to the physical nanowires, such that the defective junctions are bypassed. They use a similar approach as we do here (Chapter 5). First they construct a graph from physical nanowires and nanowire logic. Then they find a matching from the nodes of a graph that represents nanowires logic to the physical nanowires. They demonstrate their result for different memory sizes (k= 5, 6, 7, 8, and 9) and for different defect rates as high as 20%.

Another work has been done by Snider *et al.* [4]. They have demonstrated a defect-tolerant programmable logic crossbar. The crossbar architecture consists of two programmable P-FET and N-FET arrays, and two programmable switch arrays. They assume a random distribution of defects in the arrays, with the defect rates from 0% to 20%. The maximum fanin size is also an input parameter of their approach. The experimental results are done on a single application with the maximum fanin sizes of 4, 6, 8 and 10. The programming is done using a pruned exhaustive search algorithm.

1.2 Overview

In this work, we show how we can use programmable crosspoint arrays which have defective crosspoint junctions in the construction of nanoPLAs. With the techniques in this paper, we show that arrays with a 20% crosspoint defect rate are still usable with modest (13% including fanin bounding) overhead. That is, despite the fact that no rows or columns are free of defective junctions, we can still make use of more than 87% of the nanowires.

Due to the high defect rate, the number of defected junctions per array can be very large. In order to make use of the defective array by bypassing the defective junctions, this defect mapping must be applied on a per-array basis. That is, each nanoPLA will have a unique fault pattern. Since nanoPLAs are a few microns tall and 10–20 microns wide [2], we can easily have millions of these nanoPLAs on a modest die. Consequently, it is important that we minimize the time required to map around defects. To this end, we introduce a linear-time, greedy mapping algorithm for assigning logical P-terms to physical nanowires avoiding defective junctions in a fabricated nanoPLA.

Novel contributions of this work include:

- Formulation of defective crosspoint mapping problem for nanoPLAs
- Introduction of simple, greedy algorithm for linear-time mapping around defects
- Analytical identification of bounds on P-term fanin driven by array size and fault rate
- Analytical estimates and empirical characterization of mapping times
- Empirical and analytical characterization of mapping overhead for our proposed algorithm

In the next chapter, we review the emerging, bottom-up fabrication techniques for nanowires and crosspoints (Sections 2.1 and 2.2). The architectural building blocks for restoration will be described in Section 2.3. The architectural description of the address decoder to address the nanowires at the nanoscale pitch from lithographic scale wires comes in Section 2.4. We then review the conventional PLA architecture along with the nanoPLA architecture (Section 3.1 and Section 3.2). In Chapter 4, we introduce our defect model which are *breaks* and *defective junctions*. In this chapter we talk about the origin of the defects and a possible strategy to detect them.

Chapter 5 formulates the problem and introduce the basic idea for the solution. It include an overview on programming a logic plane. It also covers the challenges of programming while trying to bypass defective junctions. The mapping problem is formally stated as a bipartite graph matching problem. The logical program of nanowires are the nodes of one side of the bipartite graph and each physical nanowire (with its specific defect configuration) represents a node in the other side of the graph.

Chapter 6 reviews exact algorithms to solve the identified mapping problem and develops our linear-time heuristic algorithms. We even improve the mapping time of the algorithm further by exploiting a stochastic approach.

In Chapter 7, we analyze the algorithms based on expected case behavior for the running time and area overhead estimation. To improve the running time and area overhead we derive bounds on the size of the input fanin in Chapter 8. Chapter 9 provides experimental results which ground and confirm the analysis.

Chapter 2

Substrate

This section provides an overview of the nanotechnology substrates and some of the technology processes.

2.1 NanoWires

One-dimensional nanostructures, such as semiconductor nanowires are promising building blocks for nanoscale device applications. Different fabrication processes for synthesizing nanowires have been demonstrated. Two of them will be briefly explained in this section.

A method for synthesis of semiconductor nanowires has been developed using nanocluster catalysts of gold. This method deposits nanowire materials on the gold nanocluster catalysts. The deposition happens only through the gold nanocluster catalyst and that is how nanowires grow in one direction. This method is used to grow nanowires in one direction by Vapor-Liquid-Solid (VLS) growth process [7]. The diameter of the nanowires can be controlled at the nanometer scale by the diameter of the gold particles. By controlling the mix of elements in the environment during growth, semiconducting nanowires can be doped to control their electrical properties [8]. The doping profile along the length of a nanowire can be controlled by varying the dopant level in the growth environment over time [9]; as a result, our control over growth rate allows us to control the physical dimensions of these features down to almost atomic precision.

Having VLS grown semiconductor nanowires as a core, a doped silicon (*e.g.* boron doped, p-Si) or an insulator (*e.g.* silicon dioxide) shell can be grown by homogeneous Chemical Vapor Deposition CVD [10]. This approach grows nanowires in radial directions. CVD approach coupled with VLS has more precise control over the fabrication and doping to synthesize higher quality electronic materials than previous methods [11]. Nanowires with 20nm thickness have been fabricated with this process with diameter variation of ±4nm among 50 nanowires [11].

The doping profile controlled along the radius of these nanowires is used to control spacing between conductors and between gated wires and control wires [10] [12].

Conduction through doped nanowires can be controlled via an electrical field like Field-Effect Transistors (FETs) [13]. This property lets us build the restoring array as will be described in Section 2.3.

Techniques have been demonstrated to align a set of nanowires into a single orientation, close pack them, and transfer them onto a surface [13] [12]. This step can be repeated and rotated by 90 degrees so that we get multiple layers of nanowires [13] [12] such as crossed nanowires for building a crossbar array or memory core.

The other successful fabrication technique is imprint lithography. Nanowires with sub-10nm feature size can be made using imprint lithography [14]. This new technique has high throughput and low cost. Imprint lithography includes little damages to sensitive circuit components, including active molecules, which are used in making programmable crosspoints (Section 2.2). Chen *et al.* have developed an inexpensive process to fabricate nanoscale devices and circuits utilizing imprint lithography, shown in [15].

A technique for fabricating aligned metal nanowires through a one-step deposition process without subsequent etching or lift-off is demonstrated in [16]. Their technique uses Molecular Beam Epitaxy (MBE) to create physical template for nanowire patterning (Figure 2.1). The template is a selectively etched GaAs/AlGaAs superlattice (Figure 2.1 step A). The wires are defined by evaporating metal directly onto the GaAs layers of the superlattice after selective removal of the AlGaAs to create voids between the GaAs layers (Figure 2.1 step C). By depositing the metal solely on the GaAs layers (Figure 2.1 step B), the wire width is defined by the thickness of the GaAs layers and the separation width by AlGaAs layers. Transfer of the metal nanowires to a silicon wafer is performed by contacting the metal-coated template to a silicon oxide surface with subsequent heating process (Figure 2.1 step D). Wires deposited with this technique were uniform and continuous over 2 to 3mm length, with very few defects. The highest density nanowire patterns fabricated is 20 Platinum 8nm nanowires at a pitch of 16nm. They show no visible defects (break or short) over stretches greater than 100μm. These metal nanowires can be translated into semiconductor nanowires by further processing.

2.2 Programmable Crosspoints

Over the past few years, many technologies have been demonstrated for molecular-scale memories. So far, they all seem to have:

1. Resistance which changes significantly between ON and OFF states
2. The ability to be made rectifying
3. The ability to turn the device ON or OFF by applying a voltage differential across the junction.

UCLA and HP have demonstrated a number of molecules which exhibit hysteresis [17] [18]. HP has demonstrated an 8×8 programmable crossbar made from one of these molecules [3]. The basic structure of

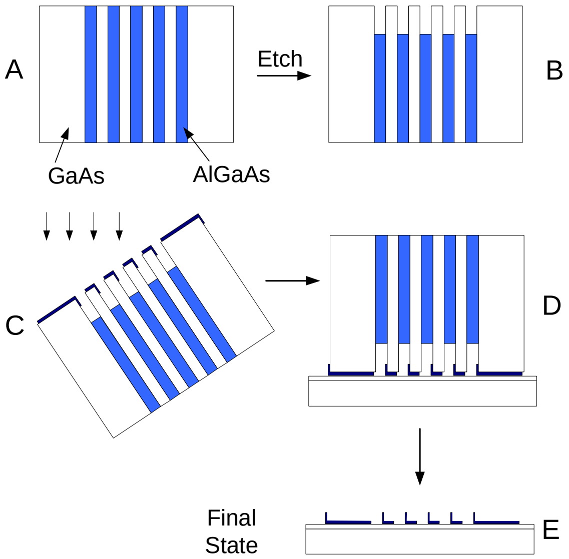


Figure 2.1: The imprint lithography process steps (Pictures modeled after [16]).

this device is a monolayer of the [2]rotaxane molecules sandwiched between two nanowires. The formation of the junction at each crosspoint acts as a reversible and nonvolatile switch. The ON and OFF resistance of their switch is $< 5 \times 10^8\ \Omega$ and $> 4 \times 10^9\ \Omega$ respectively [3].

A positive voltage of 3.5V to 7V would turn the switch ON, and a negative voltage of -3.5V to -7V would switch it OFF. A voltage with magnitude less than 3.5V would not change the resistance state [3].

The I-V characteristic of the ON-OFF state of the same programmable molecules is demonstrated in [15]. The molecular devices usually show a very high initial resistance ($> 10^8\Omega$ when measured at V=0.2V) as fabricated. This initial high resistance state is stable for $|V| < 2V$. Exceeding these voltage limits usually causes an irreversible transition to a smaller resistance. The initial resistance can be as high as $6.1 \times 10^8\Omega$, but after sweeping the voltage bias cycle from 0 to +5V, the resistance subsequently dropped to $4.3 \times 10^5\Omega$. After this step, the device becomes a reversible switch with lower cycling voltages from 0 to ±2V. After the initial step the molecular device is set in the OFF state, the I-V characteristic measured at $\pm 0.2V$ shows a resistance of $8.1 \times 10^6\Omega$. A positive voltage bias cycle between 0 to 2V is next applied to the device, and turns it ON. The resistance in this ON state is $4.8 \times 10^5\Omega$. A subsequent negative voltage bias cycle from 0 to -2V brings the device to the OFF state again with resistance of $9.2 \times 10^6\Omega$.

This ON/OFF switching cycles are repeatable. However the ratio of the ON/OFF resistance decreases after 40 switching cycle.

2.3 Nonprogrammable Restoring Crosspoints

With diodes alone we cannot invert signals which will be necessary to realize universal logic. Further, whenever an input is used by multiple outputs, diode junctions divide the current among the outputs; this cannot continue through arbitrary stages as it will eventually not be possible to distinguish the divided current from the leakage current of an OFF crosspoint. The diode junction may further provide a voltage drop at every crosspoint such that the maximum output high voltage drops at every stage.

The limitations of diode logic is overcome by inserting rectifying field-effect stages between diode stages [2]. As noted above, a doped nanowire can be gated like a Field-Effect Transistor. If the input field allows conduction, the nanowire will allow a source voltage to flow through the gated junction; otherwise conduction is cut off and the output is isolated, through a high impedance junction, from the supply. When we place a field-effect buffer or inverter on the output of a diode OR nanowire, the entire OR stage is capacitively loaded rather than resistively loaded.

The OR stage simply needs to charge up its output which provides the field for the field-effect based restoration stage. When the field is high enough (low enough for p-type nanowires) to enable conduction in the field-effect stage, the nanowire will allow the source voltage to drive its output. The field-effect stage provides isolation as there is no current flow between the diode stage and the field-effect stage output. It has been shown that nanowire field effects have good enough thresholds that we can get gain and logic level restoration with these stages [19].

2.4 Addressing Nanowires From Lithographic Scale Wires

The preceding technologies allow us to pack nanowires at a tight pitch into crossbars with programmable crosspoints at their junctions. The pitch of the nanowires can be much smaller than our lithographic patterning. The crosspoint programmability will be used to configure logic functions into these nanoscale devices. In order to make this possible, a way to selectively place a defined voltage on a single row and column wire is necessary to set the state of the crosspoint. By constructing nanowires with doping profiles on their ends, using the techniques of Section 2.1 [9] [19], each nanowire can have an address (See left end of Figure 3.2). The dimensions of the address bit control regions can be set to the lithographic pitch so that a set of crossed, lithographic wires can be used to address a single nanowire. If all the nanowires along one dimension of an array have suitably different codes, we can get unique nanowire addressability and effectively implement a demultiplexer between a small number of lithographic wires and a large number of nanowires. It cannot be controlled exactly which nanowire codes appear in a single array or how they are aligned, but if nanowires are randomly selected from a sufficiently large code space, with very high probability (over 99%) they will have unique codes. The addresses do not have to be entirely unique for this application; allowing a little redundancy will allow us to use a tighter code space. The basic stochastic addressing scheme is developed in detail in [19]. Calculations allowing redundancy are summarized in [20].

Chapter 3

Architecture Model

In this Chapter we briefly explain the nanoPLA architecture introduced in [2]. For better understanding of nanoPLA architecture, we will first show the architecture of conventional PLA and then move on to the nanoPLA architecture.

3.1 Conventional PLA Architecture

The Programmable Logic Array (PLA) is one of the first programmable hardware devices [21]. A PLA computes the sum of product function of the input signals. It has a very simple and regular architecture (Figure 3.1).

The PLA consists of two programmable NOR planes. The inputs of the device enter the first plane (AND plane) and produce the PRODUCT terms in the sum of product functions. The PRODUCT terms are called P-terms for short. The P-terms enter the second plane (OR plane) and generate the output signals.

Both AND and OR planes are programmable. The programmability is the result of the programmable junctions. Each junction is made of an NMOS transistor whose gate is controlled by a fuse or SRAM (enlarged part in Figure 3.1). If an input signal is part of a P-term function in the AND plane, the control SRAM or fuse will stay connected and if it is not part of the P-term function the SRAM or fuse will disconnect the input. The P-terms are pulled up weakly with a static load. If one of the inputs of the P-term is high it pulls down the P-term with the NMOS transistor at its junction (Figure 3.1).

The OR plane is implemented exactly as the AND plane, and it can be programmed the same way as AND plane. The inputs of the OR plane are the P-terms and its outputs are the primary outputs of the design.

Note that the AND plane actually implements AND functions of the inverted inputs rather than the original inputs. So by DeMorgan's law it is basically a NOR function. But as the inputs are available both in inverted and original form, the AND function can be implemented using the inverted inputs.

The OR plane implements the NOR function of P-terms, in the same way as the AND plane. However the inverter at the final outputs, make the overall function of the plane, an OR function.

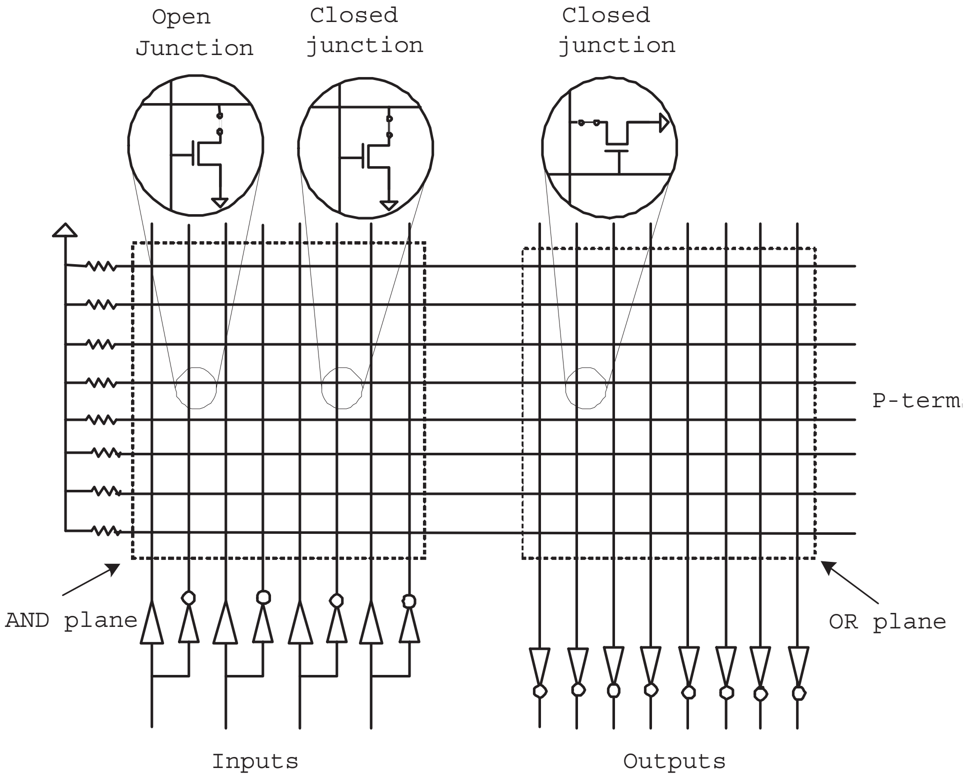


Figure 3.1: PLA Architecture

3.2 NanoPLA Architecture

The nanoPLA architecture described here is the architecture suggested in [2]. NanoPLAs, like conventional PLAs, consist of two programmable NOR planes. Figure 3.2 illustrates the two NOR planes of the nanoPLA. The first NOR plane is comparable to the AND plane in the conventional PLA architecture and the second NOR plane is comparable to its OR plane. Each NOR plane consists of two arrays: logic array and buffer/inverter array (Figure 3.2). The primary inputs enter the buffer/inverter array of the first NOR plane, then enter to the logic array of the same plane. The signals at this stage are the OR function of the buffered/inverted input signals. The OR functions enter the second NOR plane. They pass through the buffer/inverter array and then enter the logic array.

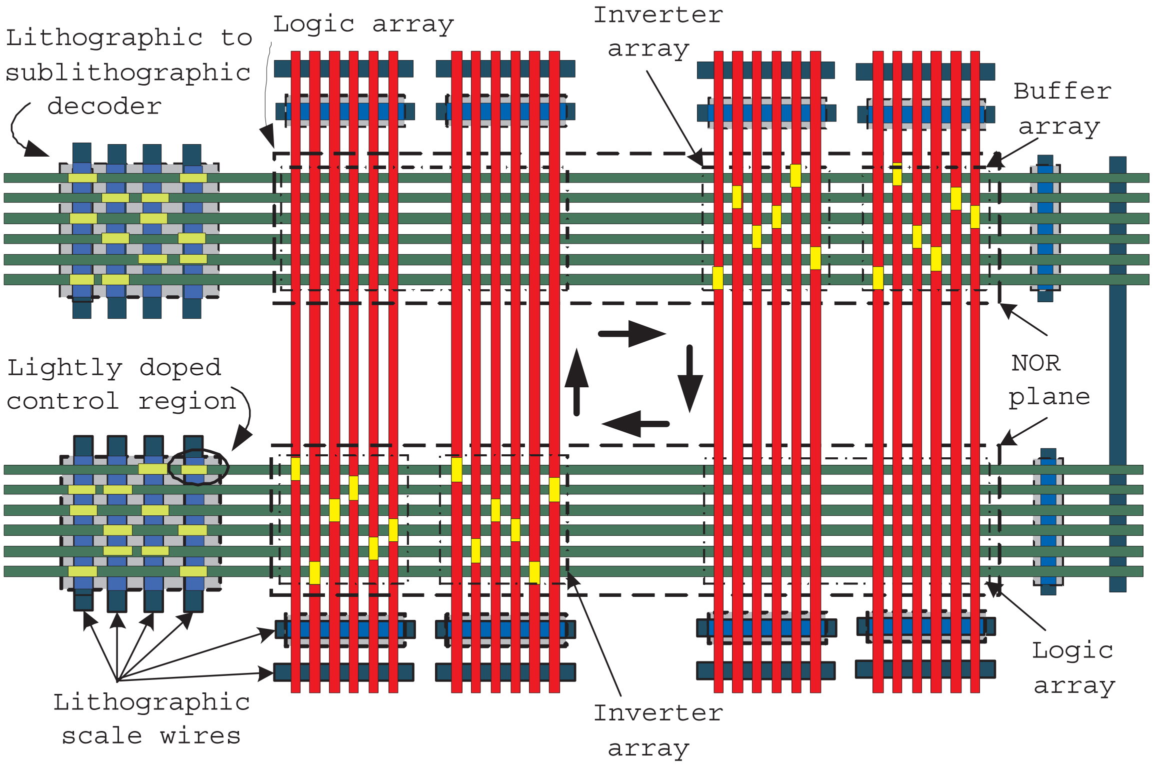


Figure 3.2: NanoPLA Architecture introduced in [2].

If the mapped logic is a 2-level logic then the primary outputs are ready at this stage. For more than 2-level logic, for example a 4-level logic, the outputs of the second NOR plane rotates back to the first NOR plane and then pass through the first and second plane one more time to produce the primary outputs. More information on how multi-level logic can be implemented on a limited number of NOR plane nanoPLA can be found in [2].

3.2.1 Logic Array

The logic array is the programmable part of each NOR plane. Its junctions are the bistable crosspoints described in Section 2.2. If the input nanowires of the array (the red vertical nanowires in Figure 3.2) are P-type doped and the output nanowires (the green vertical ones in Figure 3.2) are N-type doped then a closed crosspoint is a PN junction and each output nanowire implements the wired-OR function of its inputs. This is the reason we call them OR-term nanowires.

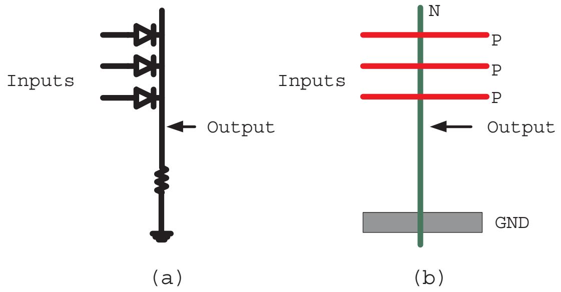


Figure 3.3: Implementation of an OR function with diodes. The output line is weakly pulled down, therefore a high signal from any of the inputs can pull it up.

The output of each OR-term is pulled down weakly. If any of the inputs is high, then it pulls up the OR-term output. Figure 3.3 shows how each of these OR-term is implemented. Figure 3.3(a) shows the circuit view of an OR-term and 3.3(b) shows a wire connections similar to Figure 3.2.

If an input participates in an OR function, the junction of that input and the OR-term nanowire representing that function will be programmed “closed”; and it will be left “open” when the input is not in the OR function. The junctions are initially in the “open” state. To program a nanoPLA some of the logic array junctions need to be programmed “closed”. To program a junction “closed” or back to “open” high voltages will be applied to nanowires crossing the junction (See Section 2.2). The decoders explained in Section 2.4 will be used to select the two nanowires crossing the junction individually.

3.2.2 Buffer/Inverter Array

The second part of the NOR plane is the buffer/inverter array. The buffer/inverter array restores the input signals using the restoring, nonprogrammable junctions (Section 2.3). The OR-term signals or the primary inputs can be selectively inverted or buffered in this array.

The inputs of the buffer/inverter array are the OR-term nanowires or the primary input nanowires, and its outputs are the restored signals. The restored signals have either the original polarity or the inverted polarity. Consequently the input signals to the logic array is either buffered or inverted. The polarity of the outputs of the buffer/inverter array will be determined at fabrication time. Some of the signals are only buffered some are only inverted and some are both buffered and inverted [2].

Our mapping algorithm is designed for the architecture described in this chapter. However this algorithm is mostly focus on the logic array and changes in the rest of the architecture does not effect it. For example our mapping algorithm is applicable to the architecture of [22] despite the architectural differences between [22] and [2].

Chapter 4

Defect Model

In this section we discuss possible defects in the nanoPLAs, and the defect model used in this work. The procedure to discover these defects will also be explained. The two more probable defects that we focus on here are:

1. Breaks in nanowires
2. Defects in programmable crosspoints

4.1 Breaks

The broken nanowires can be easily detected with the procedure suggested in [2]. By applying a voltage to only one nanowire through the decoder, one can read the value form the other end and determine if the nanowire conducts across its length. The horizontal nanowires can be tested by applying a voltage through the decoder from one end and read back the value from the other end. To test the vertical nanowires the high voltage is applied to a horizontal nanowire through the decoder, then the signal controls a vertical nanowire current through the FET-like restoring junction of the buffer/inverter array. The value of the vertical nanowire can be read from the other end. More detailed information about testing for broken wires can be found in [2].

According to their results, the time required to test the nanowires of each array is linear in the code space size of the stochastic address decoder. After detecting the broken nanowires as explained above, the only source of defects will be defective programmable crosspoints.

4.2 Defective Crosspoints

Defects in programmable crosspoints are due to the structure of the junctions. As explained in Section 2.2, the structure of a programmable crosspoint is a sandwich of bistable molecules, between two layers of nanowires. In each crosspoint there are only a few molecules. Nanowires with 40nm diameter used in [3], have the cross sectional area of 1600nm^2. According to their data $\sim$ 1100 programmable molecules can be placed in the cross section. Now if the nanowires of width 8nm, demonstrated in [16], are used, then the cross sectional area will be 64nm^2 which is $\frac{1}{25}$ of the first cross sectional area. So scaling the number of molecules in the cross section with $\frac{1}{25}$ yields only about 44 molecules.

The programmability of a crosspoint comes from the bistable attribute of the molecules located in the cross sectional area. If there are too few molecules at the crosspoint then the junction may never be able to be programmed "closed", or the "close" state may not have low enough resistance.

Let P be the probability that there is a single molecule on a unit of area (as small as a molecule area). The probability distribution function of the number of molecules on a cross sectional area of 44 times molecule area is:

$$\binom{44}{x} P^x (1-P)^{44-x} \tag{4.1}$$

Where x is the number of molecules in the cross sectional area. The green curve in Figure 4.1 illustrates this distribution function and the red curve illustrates its Cumulative Distribution Function with $P = 0.8$.

Let N_{min} be the minimum number of molecules which must be in the cross section so that the junction is programmable. Assuming $N_{min} = 32$ (the black line), then the probability that a junction is programmable is about 0.85. The value of P and N_{min} need to be defined by the physical parameter of the technology.

We abstract this into a simple crosspoint defect model. Crosspoints will be in one of two states:

- **programmable** – crosspoint can be programmed into both "closed" and "open" state. In the "closed" state the junction provides sufficiently low resistance to serve as an input to an OR function. In the "open" state the junction provides sufficient isolation that it does not act as an input to the OR function.
- **non-programmable** – crosspoint cannot be programmed into an adequate "closed" state, but can be set into a suitable "open" state. The crosspoint may fail to be programmable because there are not enough programmable molecules at the crosspoint. Consequently the "closed" state provides higher resistance than the designed threshold chosen for correct operation and timing of the PLA.

The programmability of a crosspoint can be checked by programming it to the closed state and reading back the results in the way explained in Section 2.2.

Crosspoints which cannot be programmed into a suitable "open" state will result in the entire horizontal and vertical nanowires being unusable. We treat these as nanowire defects rather than junction defects. Based on the physical model suggested above and discussion with physical scientists, we expect these defects which "short" horizontal and vertical nanowires to be much less likely and, consequently, believe it is reasonable to treat them as wire defects.

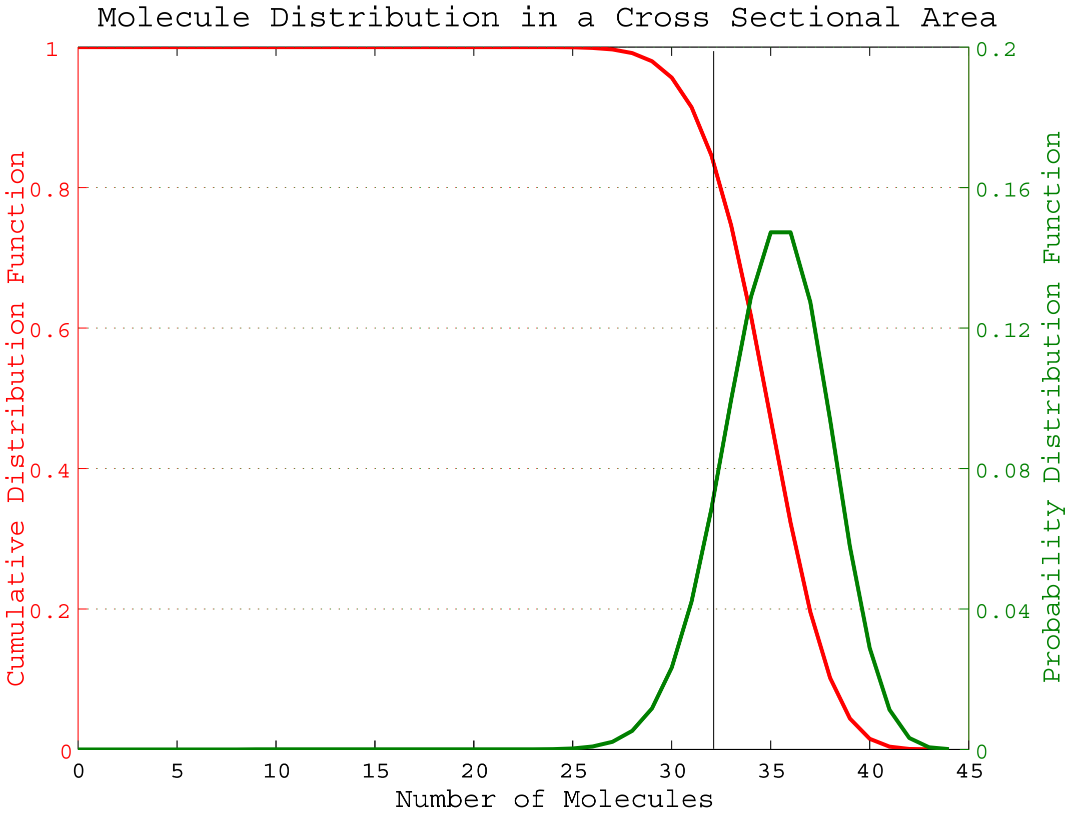


Figure 4.1: The number of molecule in a cross sectional area distribution function. The green curve is the probability distribution function and the red curve is the cumulative distribution function. The black line denotes the target minimum number of necessary molecules in the junction, 32. With $P = 0.8$, the Programmability Probability yields about 0.85.

Chapter 5

Problem Statement

In this section we first provide an overview of programming a nanoPLA. Then we discuss the challenges that programming may face due to defective crosspoints. We introduce the idea to overcome the problems of programming a defective nanowire array later in this section, and the formal problem statement comes at the end.

5.1 Overview

To implement a specific circuit on a nanoPLA, we program up the logic arrays. This means that each OR function of a design will be mapped to an OR-term nanowire. As mentioned in Section 3.2 the programmable part of the nanoPLA is the logic array. So for the rest of this paper we mainly focus on the logic array.

For clarity, we define the following terms. The *logical inputs* are the set of inputs to the OR functions. The logical inputs includes the primary inputs of the nanoPLA and the *intermediate signals* that are the outputs of the other NOR plane and rotate through the nanoPLA. In each OR function the set of logical inputs that participate in the OR function is called *ON-inputs* and those that do not participate are called *OFF-inputs*.

Henceforth we assume that the input nanowires of logic arrays are previously assigned to the logical inputs, and the order of the logical inputs is preserved. This assumption lets us use the same programming process for all the inputs whether they are primary inputs or intermediate signals. The intermediate signals are the OR-terms of the previous logic array and they may already be assigned and fixed. So in our programming operation, for simplicity, we assumed that all of the inputs are previously assigned to the logic inputs and are fixed. Figure 5.1 shows a logic array with logical inputs and OR-term nanowires.

To map each OR function to an OR-term nanowire, the crosspoints of the OR-term nanowire and the corresponding inputs of the OR function are programmed “closed”, and crosspoints of OFF-inputs are left “open”. Figure 5.2 shows an example of mapping 4 OR functions to a logic array. The logical inputs a to e

are assigned to input nanowires $H1$ to $H5$, respectively. In the cases like Figure 5.1 where there is no defect in the array, each OR function can be mapped to any nanowires. Figure 5.2 shows mapping OR functions f_1 to f_4 to nanowires $V1$ to $V4$ respectively.

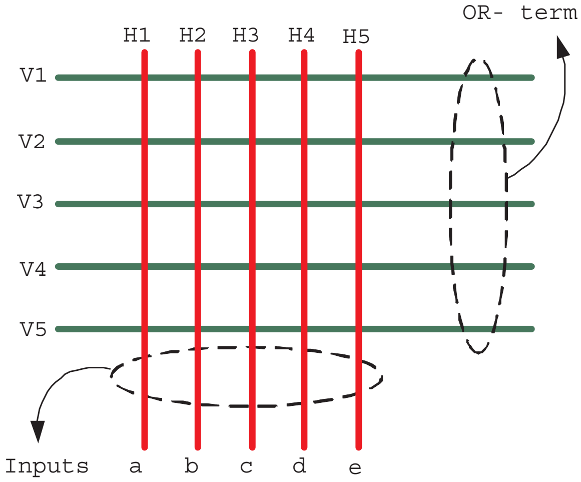
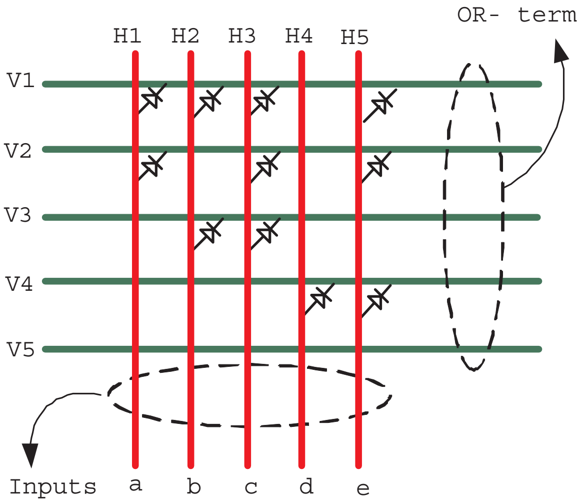


Figure 5.1: This is a logic array of a nanoPLA. The physical inputs are named H1, H2, ..., H5, and the physical OR-term nanowires are named V1, V2, ...,V5. The logical inputs a to e are assigned to the physical inputs $H1$ to $H5$.

5.2 Challenge

Logic arrays may contain some defective junctions that cannot be programmed closed, as described in Chapter 4. An OR function can be assigned to a physical OR-term nanowire if and only if each of the ON-inputs of the OR function has a corresponding programmable junction on the physical OR-term nanowire. For example if a logic array of a nanoPLA has defective junctions as marked in Figure 5.3, then the OR function $f_1 = a + b + c + e$ cannot be assigned to nanowires $w1$ anymore because of the defect in junction $(w1, c)$. It cannot be assigned to nanowire w_2 either because of the defective junctions $(w2, c)$ and $(w2, e)$. However it can be assigned successfully to nanowires $w3$, $w4$, and $w5$. Although the nanowires $w1$ cannot implement the OR function $f_1 = a + b + c + e$ it is still useful for some other OR functions, for example $f_4 = d + e$.

As the example reveals, in spite of having defective junctions in a nanowire, some OR functions can be successfully mapped to that nanowire. The challenge is to find an assignment of the OR functions to the



$$f_1 = a + b + c + e$$
$$f_2 = a + c + e$$
$$f_3 = b + c$$
$$f_4 = d + e$$

(a) (b)

Figure 5.2: (a) OR functions f_1 to f_4 are mapped to OR-term nanowires V_1 to V_4 respectively. (b) OR functions with logical inputs a, b, c, d, and e.

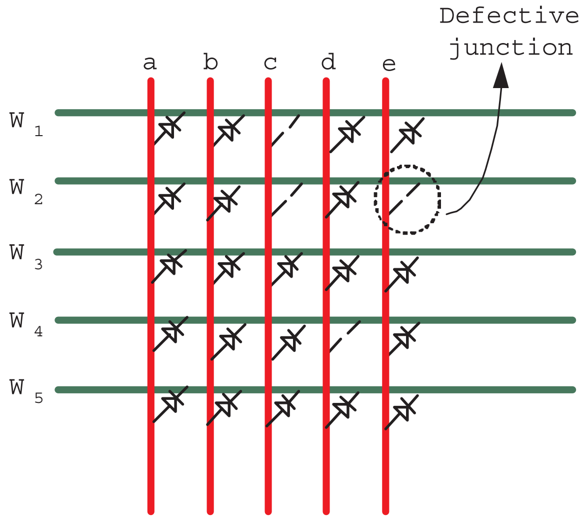


Figure 5.3: A logic array with defective(non-programmable) junctions.

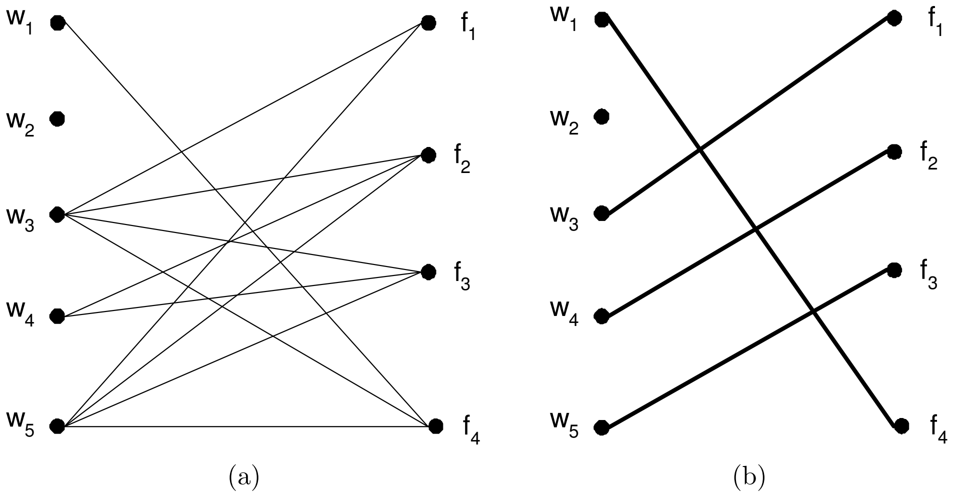


Figure 5.4: (a) The graph corresponding to OR-term nanowire of Figure 5.3 and OR-functions of Figure 5.2(b). (b) One possible matching assignment.

OR-term nanowires. Our key question is: *How do we perform this assignment with small number of spare nanowires and in small running time?*

5.3 Idea

In each OR function there are always some OFF-inputs, *i.e.* some of the junctions will always be left open. If there is a nanowire with defective junctions only at a subset of those positions, then this defective nanowire can be successfully assigned to the OR function.

Let F be the set of all the OR functions and W be the set of physical OR-term nanowires. The problem is finding an assignment of OR functions to the nanowires. This problem can be formally stated as finding a *bipartite graph matching* from the set F, the set of OR functions, to the set W, the set of physical nanowires. This will be described in more details in Section 5.4.

5.4 Formal Problem Statement

Let $f_0, f_1, ..., f_{|F|-1}$ be the set of OR functions, F, and $w_0, w_1, ..., w_{|W|-1}$ be the set of OR-term nanowires, W. If the number of inputs is N, then each OR function $f_i \in F$ is defined as:

$$f_i = (I_{i,0}, I_{i,1}, ..., I_{i,N-1}), \text{where:}$$

$$I_{i,j} = \begin{cases} 0 & \text{if input } j \text{ is an OFF-input of } f_i \\ 1 & \text{if input } j \text{ is an ON-input of } f_i \end{cases}$$

Similarly the defect configuration of each nanowire $w_i \in W$ can be defined as below:

$$w_i = (J_{i,0}, J_{i,1}, ..., J_{i,N-1}), \text{where:}$$

$$J_{i,j} = \begin{cases} 0 & \text{if crosspoint corresponding to input } j \text{ is non-programmable} \\ 1 & \text{if crosspoint corresponding to input } j \text{ is programmable} \end{cases}$$

Now we define a bipartite graph $G(F, W, E)$, where F and W are the set of nodes as defined above. For every OR function f_i in F and nanowire w_j in W, $(f_i, w_j) \in E$ if and only if:

$$\underset{0 \leq k \leq N-1}{\forall} (I_{i,k} \leq J_{j,k}) \tag{5.1}$$

Formal Definition of Bipartite Matching

In a bipartite graph $G(V_1, V_2, E)$, the set $M \subset E$ is a *matching* from V_1 to V_2 if and only if the following conditions hold:

$$\underset{u \in V_1}{\forall} \text{there is exactly one } v \in V_2, \text{ s.t.}(u, v) \in M.$$

and

$$\underset{v \in V_2}{\forall} \text{there is at most one } u \in V_1, \text{ s.t}(u, v) \in M.$$

Here $V_1 = F$, $V_2 = W$ and E is as defined above. Every matching of size $|F|$ on this bipartite graph is a valid assignment of the OR functions to the OR-term nanowires, because it finds an assignment for all of the OR functions in F and to each of them a distinct nanowire in W is assigned.

Figure 5.4(a) shows the bipartite graph $G(F, W, E)$. Set F is the set of OR functions in Figure 5.2(b), and set W is the set of defective nanowires from the logic array of Figure 5.3. Different matching assignment can be found in this graph. Figure 5.4(b) shows one possible matching.

Chapter 6

Algorithm

This section starts by showing how the bipartite graph $G(F, W, E)$ is constructed. It introduces possible exact algorithms that can solve the matching problem. Later we show how greedy heuristic algorithms can also be useful, and how the time complexity can be improved by stochastic approaches, while the quality of the results are close to exact approaches. The experimental results for the quality of the stochastic approach come later in Section 9.2.

In the time complexity explained henceforth there will be two kind of operations:

- The computing operations
- The programming and testing operations

The programming and testing operations are the operations to program a crosspoint and to read back the value. The computing operations are the operations taken to perform an algorithm. As the amount of time to perform each of the above operation is very different, for each time complexity we will identify which type of operation we are counting. *E.g.* if it takes $O(N)$ computing operations, it will be written as $O(N)_c$ and if it is $O(N)$ programming and testing operations, it will be written as $O(N)_{pt}$.

6.1 Graph Construction

To build the graph $G(F, W, E)$ the first step is to find the nodes in each one of the sets F and W. Each OR function in the design is a node in F. Marking ON and OFF inputs of each OR function, *i.e.* finding the values of $I_{i,k}$ for all the OR function $f_i \in F$ and all the inputs k, $0 \leq k \leq N-1$, takes $O(|F| \cdot N)_c$ operations. To find the nodes of W, which is the set of all unbroken nanowires, one should check the programmability of all the junctions of each OR-term nanowire, *i.e.* setting the values of $J_{j,k}$ for all the OR-term nanowires $w_j \in W$ and all the input nanowires k, $0 \leq k \leq N-1$, This takes $O(N \cdot |W|)_{pt}$ operations.

The next step is to find the set of edges, E. To make the set E, the condition (5.1) will be checked for each pair of (f_i, w_j). Checking it once takes $O(N)_c$, and checking it over all of the pairs takes $O(N \cdot |F| \cdot |W|)_c$.

So the total time complexity of the graph construction is $O(N \cdot |F| \cdot |W|)_c + O(N \cdot |W|)_{pt}$.

6.2 Exact Algorithm

If the size of the maximum matching is less than $|F|$, then there is one or more OR functions $f_i \in F$ that has no OR-term nanowire assigned to it. This means the design cannot be mapped to the physical nanoPLA. For now assume that W is large enough so that there exists a maximum matching of size $|F|$. Later in Chapter 7 we calculate how large W should be in practice.

There are a number of exact algorithms to solve the maximum bipartite matching problem. Such as the algorithm based on Ford-Fulkerson maximum flow network algorithm [23] with time complexity $O\left(|V| \cdot |E|\right)_c$. Hopcroft and Karp found a faster algorithm in 1973 with time complexity $O\left(\sqrt{|V|} \cdot |E|\right)_c$ [24]. Later Alt *et al.* [25] introduced an algorithm which is an improved version of Hopcroft-Karp, with time complexity $O\left(|V|^{1.5}\sqrt{\frac{|E|}{\log|V|}}\right)_c$.

In this section we calculate the time complexity of the last exact algorithm, which is the fastest one among the exact algorithms, in the design parameters, $|F|$ and $|W|$. Let P_J be the probability that a junction is programmable, and c_i is the number of ON-inputs in the OR function f_i. Henceforth we assume that P_J is an independent identically distributed (iid) variable. Therefore probability that $(f_i, w_j) \in E$ is ${P_J}^{c_i}$. This is the probability that the OR function f_i can be assigned to the output nanowire w_j; which means each of the c_i ON-inputs have a corresponding programmable junction in w_j. Multiplying it with the number of nanowires, $|W|$, gives the expected number of nanowires that f_i can be mapped to. In other word it is the expected node degree of f_i in the graph. Therefore the expected value of the node degree of $f_i \in F$ can be written as:

$$\mathrm{E}(D(f_i)) = |W| \cdot {P_J}^{c_i} \tag{6.1}$$

Consequently the *expected value* for the number of edges, $|E|$, is

$$\sum_{i=0}^{|F|-1} |W| \cdot {P_J}^{c_i} = O(|F| \cdot |W|)$$

Based on that, the best exact algorithm and graph construction takes

$$O\left((|F| + |W|)^{1.5}\sqrt{\frac{|F| \cdot |W|}{\log\left(|F| + |W|\right)}}\right)_c + O\left(N \cdot |F| \cdot |W|\right)_c + O\left(N \cdot |W|\right)_{pt}$$

To reduce the total time complexity, it is not enough to reduce the matching time complexity. Note that if $|F| \approx |W| \approx N$ then the exact algorithm has time complexity of:

$$O\left(\frac{N^{2.5}}{(\log N)^{0.5}}\right)_c \tag{6.2}$$

verses the construction time complexity of

$$O(N^3)_c + o(N^2)_{pt} \tag{6.3}$$

So to get any improvement it is more critical to reduce the graph construction time than the matching time.

6.3 Why do we want to improve the Running Time?

As mentioned earlier in Section 4.2 the defect probability can be very high, *e.g.* 80% defect rate is quite possible. With this defect rate in a 100 × 100 array we expect at least one defect per row or column. This shows that the number of programmable junctions common between two different array can be very small. So one-time mapping algorithm for all different arrays with different defect configurations may not be a possible solution. Now that we need to do per array based mapping, the running time of the algorithm will matter. In the next section we show how a heuristic approach reduces the total time complexity by reducing the matching time and eliminating the need to build the graph.

6.4 Greedy Heuristic Algorithm

There are heuristic algorithms that, with high probability, and small time complexity find the maximum matching. A general heuristic algorithm will be like the code in Figure 6.1.

```
While $F$ is not empty
    Choose a node $f_i \in F$
    While ($f_i$ is not matched) & ($W$ has $not\text{-}visited\text{-}by\text{-}f_i$ vertex)
        Choose a node $w_j \in W$
        If $(f_i, w_j) \in E$
            Mark$(f_i, w_j)$ as match,
            Remove $f_i$ from $F$ and $w_j$ from $W$
        Else
            Set $w_j$ $visited\text{-}by\text{-}f_i$
    EndWhile
EndWhile
```

Figure 6.1: The general strategy for heuristic algorithm.

```
1  While $F$ is not empty
2      Choose a least degree node $f_i \in F$
3      While ($f_i$ is not matched) & ($W$ has $\textit{not-visited-by-}f_i$ vertex)
4          Choose randomly a node $w_j \in W$
5          If $(f_i, w_j) \in E$
6              Mark$(f_i, w_j)$ as match,
7              Remove $f_i$ from $F$ and $w_j$ from $W$
8          Else
9              Set $w_j$ $\textit{visited-by-}f_i$
10     EndWhile
11 EndWhile
```

Figure 6.2: The framework of the heuristic algorithm used in this work.

We distinguish the different heuristic algorithms by the way they choose the nodes in lines 2 and 4 of Figure 6.1. One way is to choose both f and w randomly. Another way is to choose each of them in decreasing order of node degree. A combination of the above is another option. We obtain our best results by choosing the least degree f from F and choosing w randomly (Figure 6.2). This algorithm has the potential to run faster than the exact algorithms on average, but as the time complexity of building the graph remains:

$$O(N \cdot |F| \cdot |W|)_c + O(N \cdot |W|)_{pt} \tag{6.4}$$

the total time complexity is still high.

6.5 Stochastic Approach

Here we show how we can eliminate the need to actually build the graph $G(F, W, E)$ in order to improve the time complexity. There are two points in the algorithm of Figure 6.2 that are dependent on the graph:

1. Line 2: When we choose f_i's based on their degrees; we need to sort them first, and to do that we need to have the graph constructed.
2. Line 5: It checks for the matching condition, by checking the existence of the edge (f, w).

Instead of ordering nodes in F based on their degree, the nodes can be ordered base on their *expected value* of their degree. The *expected value* of degree of f_i, as calculated in Equation (6.1), is $({P_J}^{c_i} \cdot |W|)$. The ordering of F based on the value of $({P_J}^{c_i} \cdot |W|)$ is the same as ordering it based on the value of c_i. Consequently we can sort the set of OR function only based on the number of their ON-inputs rather than their degree in the graph.

This is intuitively true, because the OR functions with larger fanin, c_i, are harder to map, so we try to map them first.

To test the condition of line 5 of Figure 6.2, in the case that there is no graph, we need to program and test every single nanowire that is picked up to be assigned to each OR function f_i. The time complexity of programming and testing is $O(c_i)$, the fanin size of the OR function, for each OR function f_i. (Note that in order to have time complexity of $O(c_i)$ instead of $O(N)$ the $I_{i,k}$'s need to be stored efficiently(sparsely).) Hence by paying this cost; there is no longer a need to build the graph $G(F, W, E)$, the total time complexity is only due to the matching algorithm. Figure 6.3 shows the pseudocode for this algorithm.

```
Sort the elements in $F$ in decreasing order of $c_i$
While $F$ is not empty
    Choose the first $f_i \in F$
    While ($f_i$ is not matched) & ($W$ has $not\text{-}visited\text{-}by\text{-}f_i$ vertex)
        Choose a random $w_j \in W$
        If ( $\underset{I_{i,k}=1}{\forall}$ k, ($J_{j,k} == 1$)) \* try programming all the
                                                                  $c_i$'s crosspoints*\
            Mark($f_i, w_j$) as match,
            Remove $f_i$ from $F$ and $w_j$ from $W$
        Else
            Set $w_j$ $visited\text{-}by\text{-}f_i$
    EndWhile
EndWhile
```

Figure 6.3: This algorithm chooses the OR function nodes in decreasing order of the size of their ON input set and the nanowires node randomly.

Chapter 7

Analysis

In this Chapter we compute the running time complexity of the greedy algorithm both in worst case and in average case. We also estimate the area overhead for the average case.

7.1 Running Time Complexity

We first compute the worst-case time complexity. The sorting operation in line 1 takes $(|F|\log(|F|))$ (Later we show how this can be implemented in linear time using *radix sort*). As explained above, line 6 of the algorithm in Figure 6.3 takes $O(c_i)_{pt}$ running time. The maximum number of iterations of the line 4 loop is the total number of unmatched nanowires which is $W - i$. This makes the time complexity of mapping the $(i-1)$th OR function:

$$(W - i) \cdot c_i \tag{7.1}$$

The line 2 loop runs exactly for F iterations in order to map each of the OR functions. So the total time complexity in the worst-case is:

$$O\left(|F|\log(|F|)\right)_c + O\left(\sum_{i=0}^{i=|F|-1}\left((|W| - i)\cdot c_i\right)\right)_{pt} \tag{7.2}$$

Let c_M be the maximum of c_i's, then the above equation can be written as:

$$O\left(|F|\log(|F|)\right)_c + O\left(|F|\cdot|W|\cdot c_M\right)_{pt} \tag{7.3}$$

In Chapter 8 we show how to bound the size of c_M to a constant factor. Considering this the time complexity improves from:

$$O\left(\left((F+W)^{1.5}\sqrt{\frac{FW}{\log F + W}}\right) + (F\cdot W\cdot N)\right)_c + O(N\cdot|W|)_{pt} \tag{7.4}$$

of the exact algorithm, to $O\left(|F|\log(|F|)\right)_c + O\left(F \cdot W\right)_{pt}$ of our greedy approach. Now if $|F| \approx |W| \approx N$ then the improvement is from

$$O\left(\frac{N^{2.5}}{(\log N)^{0.5}} + (N)^3\right)_c + O\left(N^2\right)_{pt} \tag{7.5}$$

of the exact algorithm plus the graph construction to $O(N\log(N))_c + O(N^2)_{pt}$ of the greedy algorithm.

On average the number of iterations will be smaller than this. Let m_i be the number of iterations that it takes to find a match for each OR function f_i. The *expected value* of the number of matching for f_i in m_i nanowires is:

$$\text{E(Number of matching in } \ m_i) = m_i \cdot P_J^{c_i} \tag{7.6}$$

Therefore the size of the m_i for which the expected value is 1 is:

$$\begin{aligned} &\text{E(Number of matching in } \ m_i) = m_i \cdot P_J^{c_i} = 1 \\ &\Rightarrow m_i = P_J^{-c_i} \end{aligned} \tag{7.7}$$

So the average number of iterations of the inner loop is $P_J^{-c_i}$ for each f_i; and hence the total time complexity in the average case is:

$$O\left(|F|\log(|F|)\right)_c + O\left(\sum_{i=0}^{i=|F|-1}\left(P_J^{-c_i} \cdot c_i\right)\right)_{pt} \tag{7.8}$$

and replacing c_i's with c_M:

$$O\left(|F|\log(|F|)\right)_c + O\left(|F| \cdot \left(P_J^{-c_M} \cdot c_M\right)\right)_{pt} = O\left(|F|\log(|F|)\right)_c + O\left(|F|\right)_{pt} \tag{7.9}$$

If the value of c_M is small, which it is in most of the cases, then the programming and testing time complexity is basically linear with the value of $|F|$. This can be observed from the experimental results in Figure 9.1. Note that when the programmability of a junction is tested in our heuristic algorithm the information will be stored for further references. So one junction will not be tested multiple times. Therefore the total number of *Program and Test* operations are much less than $\sum_{i=0}^{i=|F|-1}\left(P_J^{-c_i} \cdot c_i\right)$.

7.2 Area Overhead Estimation

Here we compute how large W, number of nanowires, should be for each design so that it can be successfully mapped to the physical nanoPLA. When matching the ith OR function we have $|W| - i$ unmatched nanowires. Applying Equation 6.1, the expected number of nanowires that OR function f_i can be matched to is $P_j^{-c_i} \cdot (|W| - i)$. This value need to be at least one. Therefore applying this for all the F OR-functions defines the following lower bound on the size of W.

$$\underset{0 \leq i \leq |F|-1}{\forall} \left((|W| - i) \cdot P_J^{c_i} \geq 1 \right) \Rightarrow \underset{0 \leq i \leq |F|-1}{\forall} \left(|W| \geq P_J^{-c_i} + i \right) \tag{7.10}$$

We can also derive tighter lower bound on the size of $|W|$. Let $P_{f_i-mapped}$ be the probability of successfully assigning f_i to a nanowire. Remember that in our algorithm the set of nanowires that f_i can choose from, is of size $(|W| - i)$. Therefore

$$P_{f_i-mapped} = 1 - (1 - {P_J}^{c_i})^{|W|-i} \tag{7.11}$$

Hence the probability of successfully mapping all the OR functions is:

$$\prod_{i=0}^{|F|-1} \left(1 - (1 - {P_J}^{c_i})^{|W|-i} \right) \tag{7.12}$$

Let Y be the yield of mapping designs to nanoPLA. Then the following inequality gives another lower bound on the size of W:

$$\prod_{i=0}^{|F|-1} \left(1 - (1 - {P_J}^{c_i})^{|W|-i} \right) > Y \tag{7.13}$$

This bound shows the tradeoff between yield Y and the area overhead $(\frac{|W|}{|F|})$. Expecting higher yield results in larger area overhead. In next section we show how the value of $\frac{|W|}{|F|}$ can be reduced by bounding the size of c_i.

Chapter 8

Fanin Bounding

In this section we show how the running time complexity and area overhead of the mapping can be improved by bounding the size of c_i's (number of ON input). We show the effect of bounding the size of c_i's by an example from the IWLS93 benchmark set [26]. In this example $|F| = 1186$, $c_M = 772$ and $P_J = 0.95$.

The lower bound on $|W|$ for mapping a single nanowire with $c_i = c_M = 772$ can be calculated by applying Equation (7.10):

$$(0.95)^{-772} \leq W \Rightarrow 10^{17} \leq W$$

This result is obviously undesirable. We can improve this result by exploiting decomposition. For example this nanowire can be decomposed into 8 nanowires, such that 7 of them have $c_i = 100$, and 1 has $c_i = 72$. Then the lower bound on $|W|$ to map all of these nanowires by applying Equation (7.10) is:

$$\begin{aligned} &\max\left(\underset{0\leq i\leq 6}{\forall}\left((0.95)^{-100}+i\right), \left((0.95)^{-72}+7\right)\right) \leq W \\ &\Rightarrow 173 \leq W \end{aligned}$$

Although fanin bounding increases the number of OR functions, the actual area overhead after mapping will be much smaller.

Bounding the fanin also improves the running time dramatically. Here we show how it improves the running time of the previous example. Based on the algorithm (Figure 6.3) the OR function with the largest c_i will be mapped first. In the unbounded case the largest c_i is 772. Applying Equation (7.1) and (7.7) results in the following number of programming and testing operations, on average, to only map the first OR function (P_J is assumed 0.95.).

$$P_J^{-c_M} \cdot c_M \Rightarrow 0.95^{-772} \times 772 \approx 10^{20} \tag{8.1}$$

Now if this OR function is divided into 8 OR functions, as suggested in the previous example, 7 of the OR functions with $c = 100$ and the last with $c = 72$, then the total time complexity to map these 8 OR functions is:

$$\sum_{i=0}^{i=6} \big((0.95)^{-100}\big) \times 100 + \big((0.95)^{-72} \times 72\big) \approx 130,000$$

The comparison of this number with the unbounded fanin OR function case shows great improvement.

Figure 8.1 shows what fraction of the OR functions in each design need to be divided to smaller fanin OR functions if the size of $|W|$ is desired to be $|F|$ or $1000 \times |F|$. The x-axis in this graph is the number of OR functions in each design, *i.e.* $|F|$. Columns of points on the x-axis is dedicated to the OR functions of a single design with $|F|$ equal to the value of x at that point. For example the highlighted points show the OR functions of a design with $|F| = 1186$. The y-value of each point is the c_i size of that OR function. The curves show the estimation in Equation (7.10) of the maximum size of c_i when $|W| = |F|$ or $|W| = 1000 \times |F|$. Considering Equation (7.10), if 0% area overhead is desired ($|W| = |F|$) in average, then:

$$P_J^{-c_M} \leq |F| \Rightarrow c_M \leq -\log_{P_J} |F| \tag{8.2}$$

Similarly the lower bounds for c_i's related to the case when $|W| = 1000 \times |F|$ will be:

$$-\log_{P_J} (1,000 \times |F|)$$

For example all the OR functions of the design with highlighted yellow diamonds have c_i's below the maximum c_i related to the case $|W| = |F|$, except two points.

These curves show that even with very large $|W|/|F|$ of 1000, some OR functions cannot be mapped and therefore the decomposition is inevitable.

From the graphs of Figure 8.1 we can observe that the number of OR functions with large ON input set is relatively small (the diamonds above the curves). Hence we only need to decompose a few OR functions to get a large benefit in area overhead and running time.

The facts that large-fanin OR functions take very long running time and large area overhead in mapping along with the fact that the number of these OR functions are relatively small suggest we should bound the fanin size. Bounding the size of c_M with $-\log_{P_J} |F|$ yields the ratio of 1 for $|W|/|F|$ on average (The F here is the number of OR functions after bounding).

The algorithm of Figure 6.3 takes $O(|F| \log(|F|))_c$ due to the time complexity of the sorting the OR functions in line 1 and the rest of the algorithm takes linear time. After bounding the fanin, the value of c_M is limited to a constant factor, so the sorting algorithm of line 1 can be replaced by *Radix Sort* algorithm [23]. This brings the time complexity of the *computing operations* of the mapping algorithm down to linear time.

This section showed how to bound the size of the fanin so that on the average case the area overhead will be zero (Equation (8.2)). As this estimation is true for the average cases there will be some cases that the fanin needs to be bounded more tightly so that the desired area overhead will be achieved. To fix this bounding per design we dynamically make the bound tighter until the desired area overhead is achieved. So instead of using Equation (8.2) we use the following Equation:

$$c_M \leq -\log_{P_J}\left(\frac{|F|}{i}\right)$$

Where it starts with $i = 1$, and if the bounding was not tight enough (larger area overhead than what is desired) then i will be incremented by 1. This allows us to pay more area overhead for fanin bounding in order to save more area for mapping. Chapter 9 shows the experimental results and they reveal that in most cases $i = 1$.

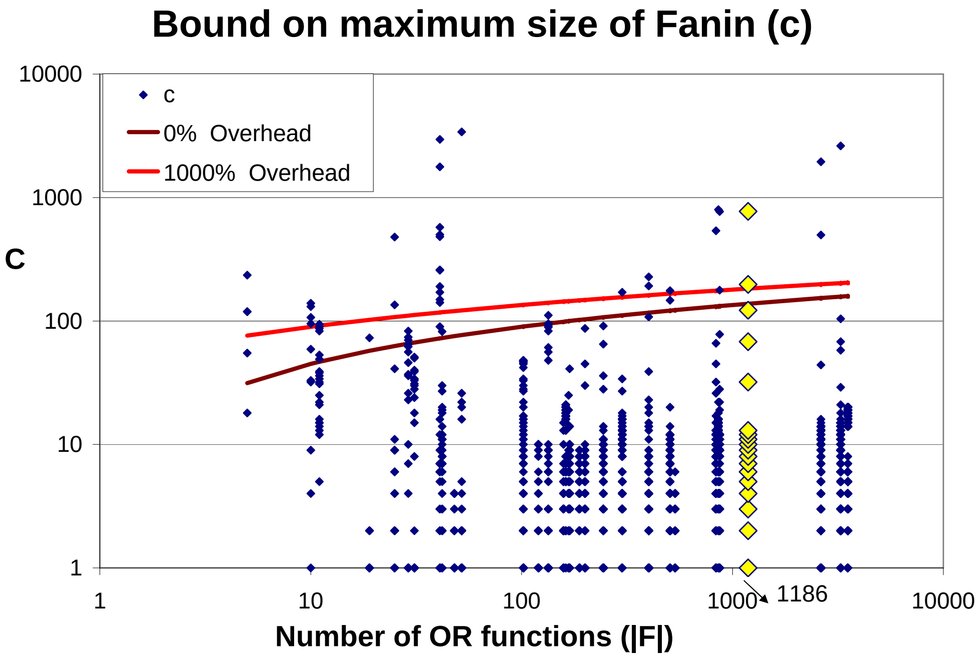
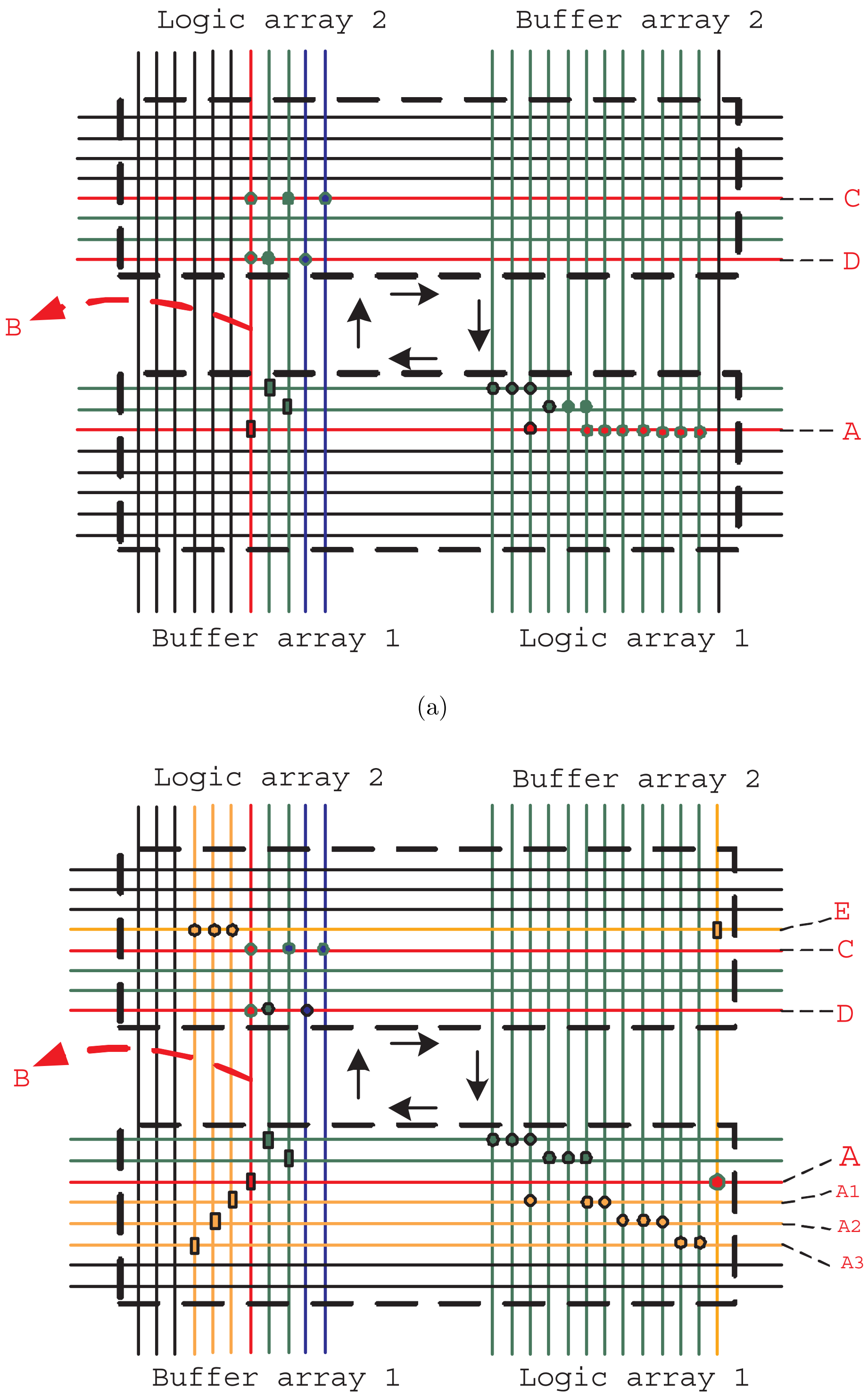


Figure 8.1: This graph shows all the c_i's of the OR functions of each design. Each curve shows the maximum size of c for each design, so that it satisfies a certain area overhead. The OR functions above the curves need to be bounded.

8.1 Bounding Procedure

Bounding the large-fanin OR functions means we break the OR function to the number of OR function with smaller c_i size.

Figure 8.2 shows how an OR function with c=8 will be decomposed into OR functions with $c \leq 3$. In Figure 8.2(a) the OR function on nanowire A in logic array 1 has c=8. It gets buffered in the buffer array 1 and makes input B to the logic array 2. The signal B is in the ON input set of two OR functions in the logic array 2, C and D. As c_M is set to be 3, the OR function A will be divided into 3 OR functions *A1, A2* and *A3* in logic array 1. They are OR-ed together in logic array 2 and make signal E. The logic of this signal is the same as logic of the original A OR function. Now that the signal A is ready at buffer array 2 rather than 1 we could shift C and D OR function from the logic array 2 to the next array of the buffer array 2 which is logic array 1. But here for simplicity we keep all of the other OR functions in their own place, and rotate the OR function E which implements the value of OR function A to logic array 1 and then to buffer array 1, which was its original position. This adds two logic level delay to the OR function A. If the size of ON inputs of signal E was more than c_M then the decomposition process would be repeated for signal E. Therefore for an OR function with c ON inputs, the decomposition process happens $\log_{c_M}(c)$ times.



(a)

(b)

Figure 8.2: (a) The original design, (b) The design with c value bounded by $c_M = 3$.

Chapter 9

Experimental Results

The mapping simulation is tested over three different benchmarks:

1. Selected elements of datapath.
2. Small examples form IWLS93 benchmark suit [26].
3. PLA book examples [27].

The total number of benchmarks is 358 designs and each design has 2 logic arrays. The designs have been first synthesized to multilevel logic and then rotated through two NOR planes of a nanoPLA [2].

The defective junctions are distributed randomly over any array. We assumed that the probability of programmability of each junction is an *iid* random variable, called P_J. We ran our mapping algorithm on the benchmark sets for different values of $P_J = 0.8$, 0.85, 0.90 and 0.95.

In order to get a valid average result, we ran our algorithm 100 times on each benchmark and averaged the results.

9.1 Running Time

Figure 9.1 shows a graph that estimates the total number of iteration to map each design. The graph shows the estimation both for bounded c and unbounded c cases. It also shows the simulation results for bounded c. The value of the x-axis is the number of OR functions in the original designs and not the number of OR functions after bounding the c size. The average number of iterations for each OR function is $P_J^{-c_i}$, so the total number of iterations in the average case is:

$$O\left(\sum_{i=0}^{i=|F|-1} P_J^{-c_i}\right) \tag{9.1}$$

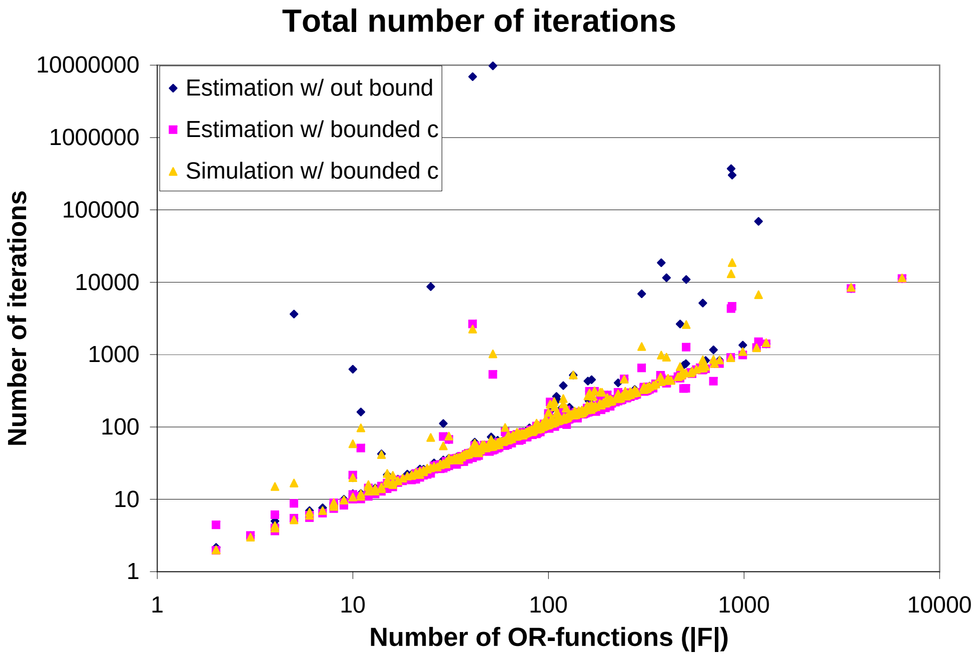


Figure 9.1: In this graph $P_J = 0.95$. It shows the total number of iterations of the algorithm to map the whole design. There are two curves showing the estimated number of iterations for designs with no bound on size of c and design with bounded c size. There is also a curve showing the simulation results with bounded c size. The value of c is bounded by $(-\log_{P_J}|F|)$.

This yields the total time complexity of

$$O\left(\sum_{i=0}^{i=|F|-1}\left(P_J^{-c_i}\cdot c_i\right)\right) \tag{9.2}$$

Figure 9.1 plots the Equation (9.1) with the value of $|F|$ both for bounded c and unbounded c cases, along with the number of iteration from the simulation results.

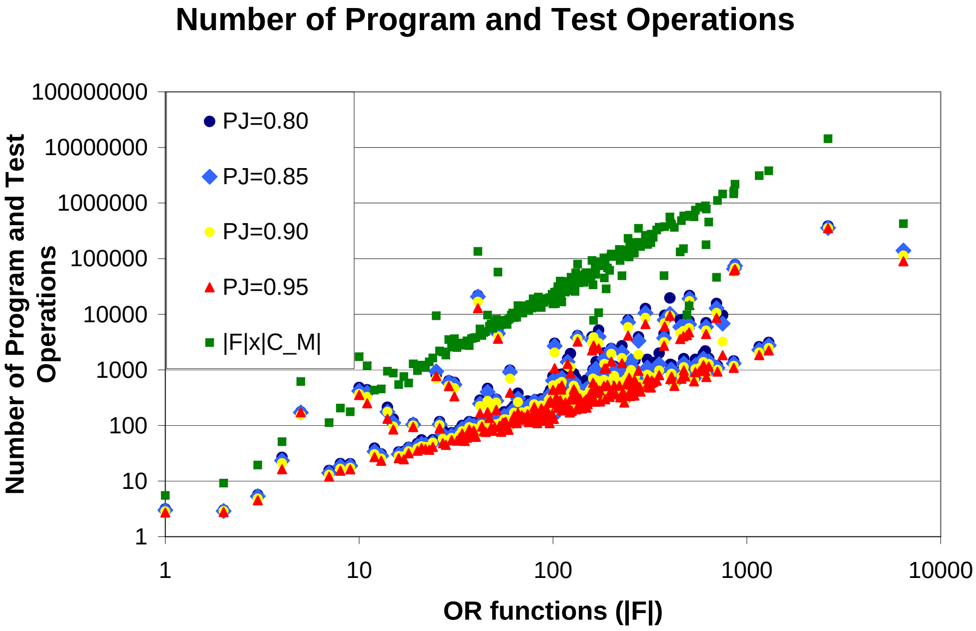


Figure 9.2: This graph plots the number of *Program and Test* operations for the exact algorithm ($|F| \cdot |c_M|$) and for the stochastic greedy algorithm for different programmability probability ($|P_J|$).

Figure 9.2 plots the number of *Program and Test* operations for the exact algorithm ($|F| \cdot |c_M|$) and for the stochastic greedy algorithm for different programmability probability ($|P_J|$).

This exactly follows the average case analysis in Section 7.1. The analysis suggests that the running time will be a linear function of the number of OR functions ($|F|$), which is supported here by the simulation results in Figure 9.2.

Note that the graph is plotted in log-log scales. As both of the axes are in log scales and the line slope is about 1 then the normal graphs will have the linear curve. The jumps on top of the linear shape is due to the designs with a few OR functions and large c.

The time complexity is improved by about two orders of magnitude (the difference of the green points from the rest of the points).

9.2 Area Overhead

Chapter 8 shows how to bound the size of the fanin so that in the average case the area overhead will be zero (Equation (8.2)). It also shows how to improve the bound specifically for each design using the below equation:

$$c_M \leq -\log_{P_J} \frac{|F|}{i} \tag{9.3}$$

Figure 9.3, 9.4 and 9.5 plot the simulation results for the area overhead. The mapping has been done 100 times for each design and the average value of the result for each design is stored. In cases that there are multiple designs with the same number of OR functions, ($|F|$), the average value of these design is shown. The bounding overhead, the mapping overhead and the total overhead curves are respectively plotted in Figure 9.3, 9.4 and 9.5. The total area overhead is the result of multiplying the bounding overhead and mapping overhead.

Figure 9.6 shows the average area overhead ratio over all the benchmark set designs. Bounding the fanin scales the number of OR functions by an average factor of 1.11 for a defect rate of 0.20. An additional factor of 1.02 in overhead is applied after physically mapping these OR functions onto nanowires, for the same defect rate. The total area overhead is the product of these two overheads, which equals 1.13.

Table 9.1 shows the value of fanin bounding for different programmability probability (P_J) values. The c_M values of this table calculated specifically for each benchmark and averaged over 100 times running (The table entries are the *floor* of the average result). The experimental results show that in most cases (more than 98%) the value of i is equal to 1 (Table 9.2).

Finally we compare the area overhead of this greedy algorithm with an exact bipartite matching algorithm. The bipartite matching algorithm is based on the network maximum flow algorithm of Ford-Fulkerson, and has running time complexity of $O(E \cdot V)$ when E is the number of edges of the graph and V is the number of vertices. The comparison is done on a 4×4 multiplier that is implemented in 4-level logic and has been rotated in two planes. The first plane has 66 inputs and 697 OR functions and the other one has 697 inputs and 25 OR functions. The simulations are done for different values of P_J. In Figure 9.7 area overhead of

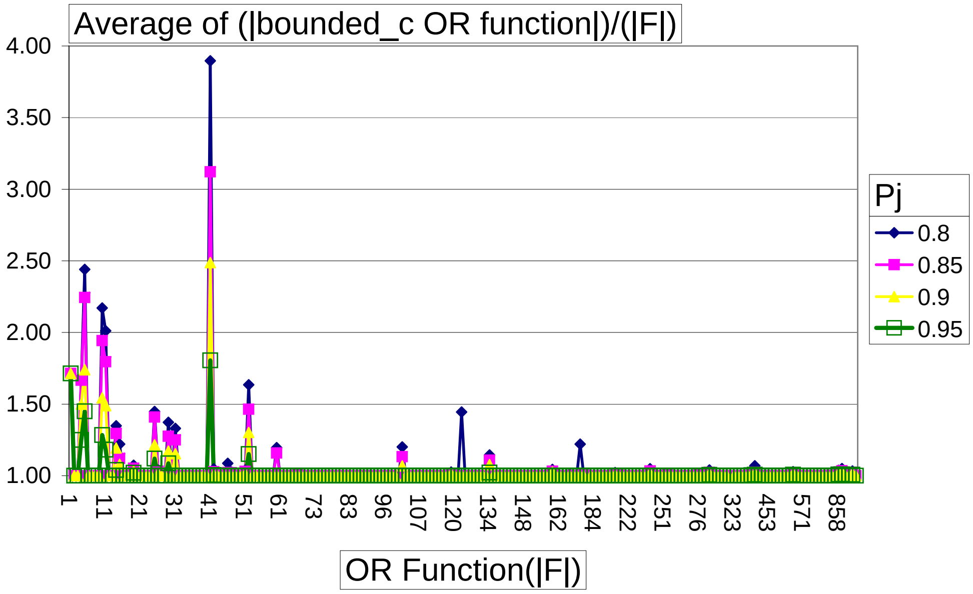


Figure 9.3: The area overhead due to bounding the fanin size is plotted for four different Programmability probability (P_J).

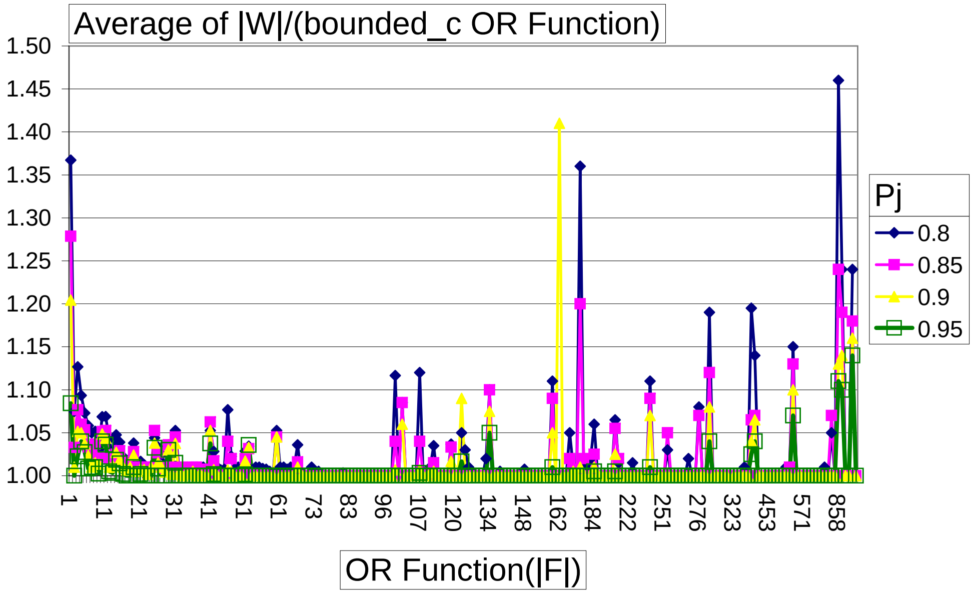


Figure 9.4: The area overhead due to mapping the bounded fanin OR functions is plotted for four different Programmability probability (P_J).

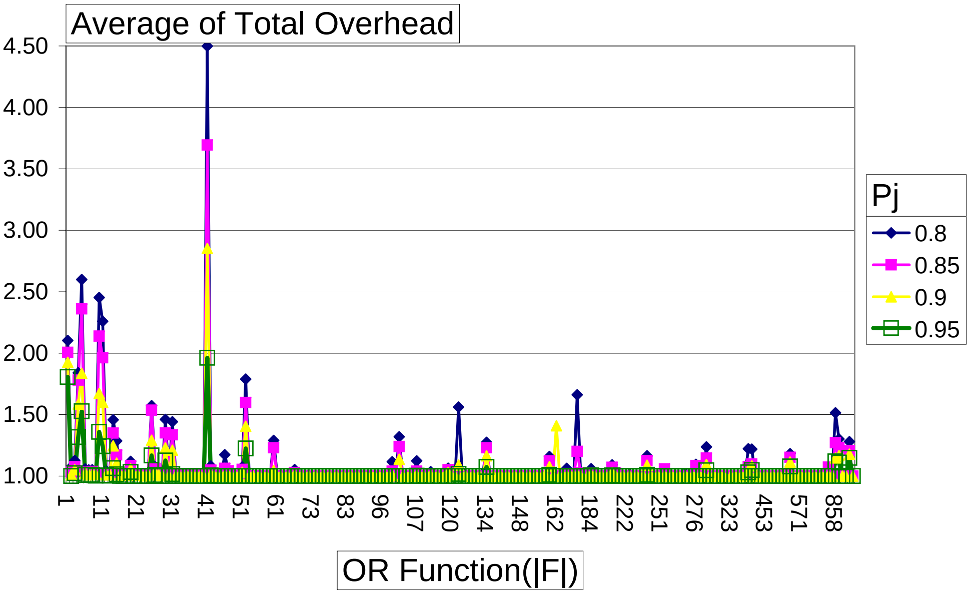


Figure 9.5: The total area overhead is plotted for four different Programmability probability (P_J).

	Fanin Bound (c_M)			
$\lvert F\rvert$	$P_J = 0.8$	$P_J = 0.85$	$P_J = 0.9$	$P_J = 0.95$
1	2	2	2	2
5	7	9	16	32
10	11	14	21	44
15	12	17	26	53
20	14	19	29	59
25	14	19	30	60
30	16	21	33	67
40	17	23	36	72
50	18	25	38	77
60	18	24	38	80
70	20	27	41	83
80	20	27	42	86
90	21	28	43	88
100	20	29	44	90
120	22	30	46	94
141	23	31	47	97
150	23	31	48	98
200	23	32	50	104
300	22	31	50	108
401	22	34	53	112
504	28	39	60	122
611	29	40	61	126
750	29	40	63	130
869	25	35	55	118
1159	32	44	67	138
1298	33	45	69	140
2625	29	40	63	136
6425	40	54	84	171

Table 9.1: Fanin bounding value for different programmability probability (c_M).

i	**Percentage**
1	98.01%
2	1.66%
3	0.20%
4	0.04%
6	0.08%

Table 9.2: This table shows that most of the time (more than 98%) the value of i in Equation (9.3) is equal to 1.

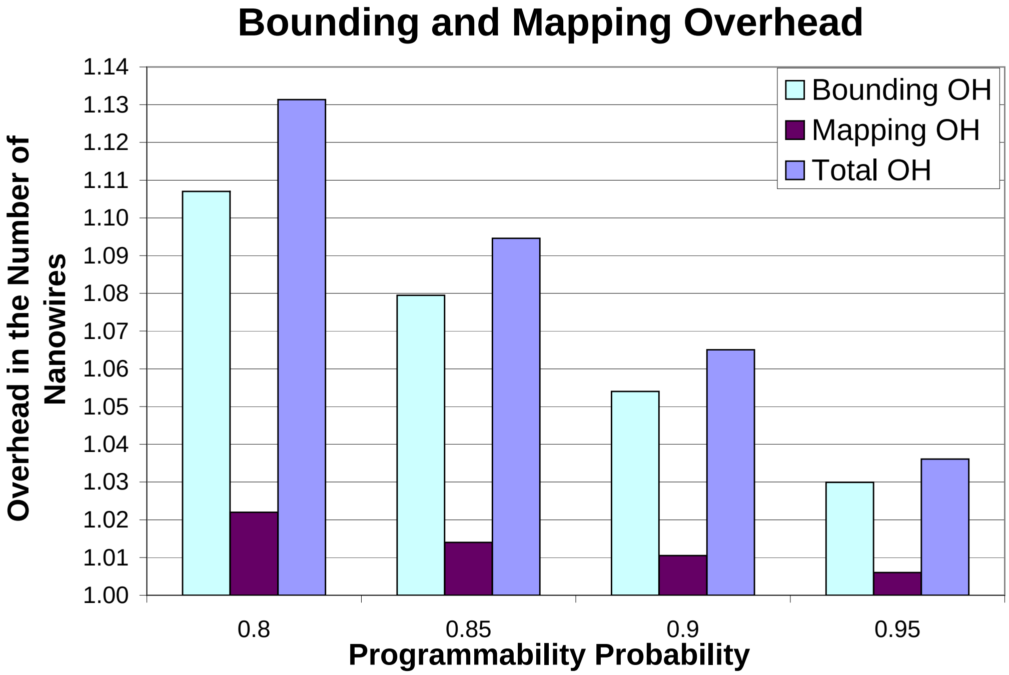


Figure 9.6: Bounding OH is the average area overhead ratio of $|F\_bounded_c|$ over $|F|$, and Mapping OH is the ratio of $|W\_bounded_c|$ over $|F|$. The ratio of the final number of nanowires over the original number of OR functions is the product of the previous two ratios.

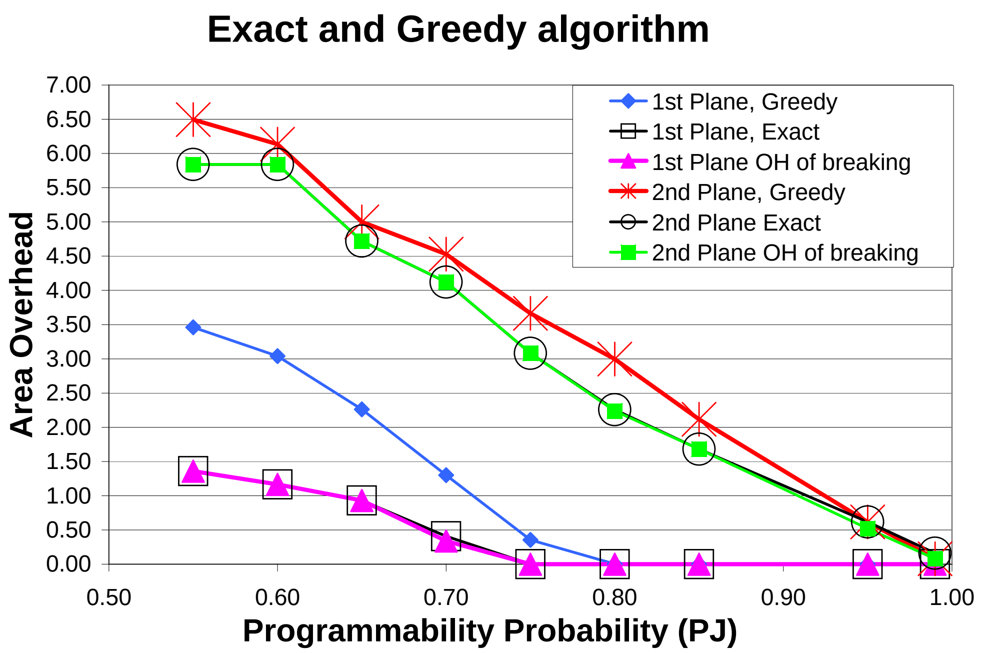


Figure 9.7: This figure compares the area overhead of an exact algorithm with our greedy algorithm for a 4×4 multiplier.

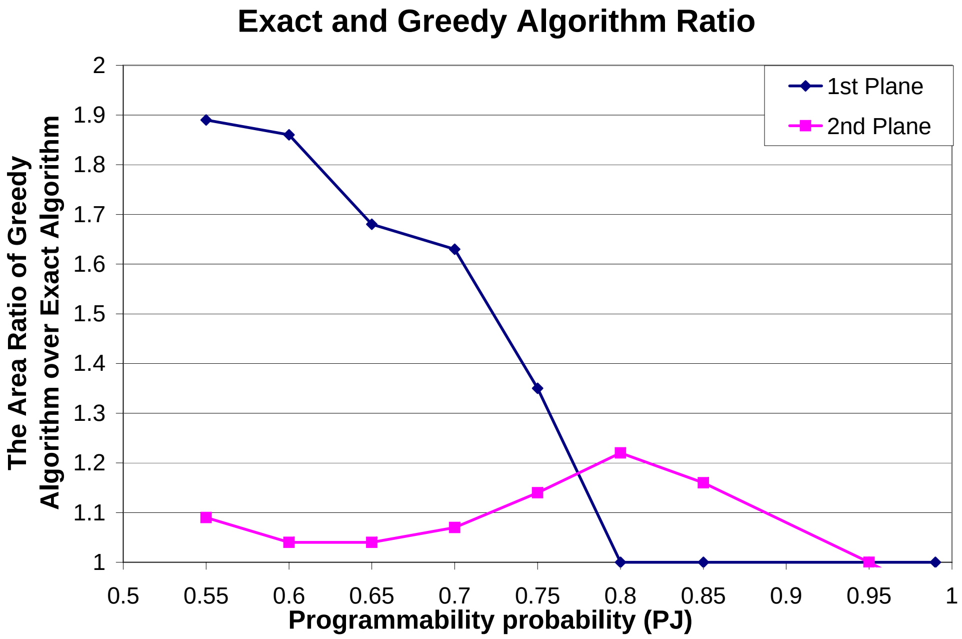


Figure 9.8: Ratio of the size of $|W|$ of greedy algorithm over exact algorithm: $\frac{|W_{greedy}|}{|W_{exact}|}$.

each of the planes is plotted for both greedy and exact algorithm. The area overhead due to breaking the large c OR functions to smaller c OR function is plotted, too.

In Figure 9.8 the ratio of the total area of the greedy algorithm over the total area of the exact algorithm is plotted. The area ratios are below 2 for the first plane, and 1.23 for the second plane even for defect probability as large as 20%. For low enough defect probability (below 20%) the area overhead of greedy algorithm to exact algorithm is 0% for the first plane and is below 20% for the second plane.

Chapter 10

Summary

A plausible architecture for nanoPLA design is suggested in [2]. The nanowires width can be built down to 3nm, making the active device area as small as 9nm^2. The defect rate of different fabrication processes is unknown but expected to be on the order of a few defects per 100 junctions. This suggests searching for an efficient programming operation that tolerates the defective devices.

Furthermore this programming operation must be fast. The reason is that due to the small size of the nanoPLAs, plenty of them can be placed on a chip and as the defect rate is high each nanoPLA will have a unique fault pattern. Therefore the defect mapping must be applied on each array separately and this suggests having a fast mapping algorithm.

In this work we compare the exact matching algorithm with a suggested greedy algorithm. Assuming that $|F| \approx |W| \approx N$, the time complexity of our algorithm is, $O(N)$ *program and test* operations and $O(N)$ *computing* operations, while the time complexity of the exact algorithm plus graph construction is $O(N^2)$ *program and test* operations and $O(N^3)$ *computing* operations.

We also showed that it is necessary to bound the fanin size in order to achieve reasonable running time and area overhead for matching. Including bounding the fanin and mapping, our algorithm can tolerate defect rates as high as 20% with an average overhead factor of less than 13%.

Bibliography

[1] S. C. Goldstein and M. Budiu, “NanoFabrics: Spatial Computing Using Molecular Electronics,” in *ISCA*, June 2001, pp. 178–189.

[2] A. DeHon and M. J. Wilson, “Nanowire-Based Sublithographic Programmable Logic Arrays,” in *FPGA*, February 2004, pp. 123–132.

[3] Y. Chen, G.-Y. Jung, D. A. A. Ohlberg, X. Li, D. R. Stewart, J. O. Jeppesen, K. A. Nielsen, J. F. Stoddart, and R. S. Williams, “Nanoscale Molecular-Switch Crossbar Circuits,” *Nanotechnology*, vol. 14, pp. 462–468, 2003.

[4] G. Snider, P. Kuekes, and R. S. Williams, “CMOS-like Logic in Defective, Nanoscale Crossbars,” *Nanotechnology*, vol. 15, pp. 881–891, June 2004.

[5] “International Technology Roadmap for Semiconductors,” <http://public.itrs.net/>, 2003.

[6] M.-H. Lee, Y. K. Kim, and Y.-H. Choi, “A Defect-Tolerant Memory Architecture for Molecular Electronics,” *IEEE Transactions on Nanotechnology*, vol. 3, pp. 152–157, March 2004.

[7] X. Duan and C. M. Lieber, “General Synthesis of Compound Semiconductor Nanowires,” *Advanced Materials*, vol. 12, no. 4, pp. 298–302, 2000.

[8] Y. Cui, X. Duan, J. Hu, and C. M. Lieber, “Doping and Electrical Transport in Silicon Nanowires,” *Journal of Physical Chemistry B*, vol. 104, no. 22, pp. 5213–5216, June 8 2000.

[9] M. S. Gudiksen, L. J. Lauhon, J. Wang, D. C. Smith, and C. M. Lieber, “Growth of Nanowire Superlattice Structures for Nanoscale Photonics and Electronics,” *Nature*, vol. 415, pp. 617–620, February 7 2002.

[10] L. J. Lauhon, M. S. Gudiksen, D. Wang, and C. M. Lieber, “Epitaxial Core-Shell and Core-Multi-Shell Nanowire Heterostructures,” *Nature*, vol. 420, pp. 57–61, 2002.

[11] A. B. Greytak, L. J. Lauhon, M. S. Gudiksen, and C. M. Leiber, “Growth and Transport properties of complementary germanium nanowire field-effect transistors,” *Applied Physics Letters*, vol. 84, pp. 4176–4178, May 2004.

[12] D. Whang, S. Jin, and C. M. Lieber, “Nanolithography Using Hierarchically Assembled Nanowire Masks,” *Nanoletters*, vol. 3, no. 7, pp. 951–954, July 9 2003.

[13] Y. Huang, X. Duan, Y. Cui, L. Lauhon, K. Kim, and C. M. Lieber, “Logic Gates and Computation from Assembled Nanowire Building Blocks,” *Science*, vol. 294, pp. 1313–1317, 2001.

[14] S. Y. Chou, P. R. Krauss, and P. J. Renstrom, “Imprint Lithography with 25-Nanometer Resolution,” *Science*, vol. 272, pp. 85–87, 1996.

[15] Y. Chen, D. A. A. Ohlberg, X. Li, D. R. Stewart, R. S. Williams, J. O. Jeppesen, K. A. Nielsen, J. F. Stoddart, D. L. Olynick, and E. Anderson, “Nanoscale Molecular-Switch Devices Fabricated by Imprint Lithography,” *Applied Physics Letters*, vol. 82, no. 10, pp. 1610–1612, 2003.

[16] N. A. Melosh, A. Boukai, F. Diana, B. Gerardot, A. Badolato, P. M. Petroff, and J. R. Heath, “Ultrahigh-Density Nanowire Lattices and Circuits,” *Science*, vol. 300, pp. 112–115, April 4 2003.

[17] C. Collier, G. Mattersteig, E. Wong, Y. Luo, K. Beverly, J. Sampaio, F. Raymo, J. Stoddart, and J. Heath, “A [2]Catenane-Based Solid State Reconfigurable Switch,” *Science*, vol. 289, pp. 1172–1175, 2000.

[18] C. P. Collier, E. W. Wong, M. Belohradsky, F. M. Raymo, J. F. Stoddard, P. J. Kuekes, R. S. Williams, and J. R. Heath, “Electronically Configurable Molecular-Based Logic Gates,” *Science*, vol. 285, pp. 391–394, 1999.

[19] A. DeHon, P. Lincoln, and J. Savage, “Stochastic Assembly of Sublithographic Nanoscale Interfaces,” *IEEE Transactions on Nanotechnology*, vol. 2, no. 3, pp. 165–174, 2003.

[20] A. DeHon, “Law of Large Numbers System Design,” in *Nano, Quantum and Molecular Computing: Implications to High Level Design and Validation*, S. K. Shukla and R. I. Bahar, Eds. Kluwer, 2004, ch. 7, pp. 213–241.

[21] C. Mead and L. Conway, *Introduction to VLSI Systems.* Addison-Wesley, 1980.

[22] A. DeHon, “Design of Programmable Interconnect for Sublithographic Programmable Logic Arrays,” in *FPGA*, February 2005, pp. 127–137.

[23] T. Cormen, C. Leiserson, and R. Rivest, *Introduction to Algorithms.* MIT Press, 1990.

[24] J. E. Hopcroft and R. M. Karp, "An $n^{2.5}$ Algorithm for Maximum Matching in Bipartite Graphs," *SIAM Journal on Computing*, vol. 2, no. 4, pp. 225–231, 1973.

[25] H. Alt, N. Blum, K. Mehlhorn, and M. Paul, "Computing a maximum cardinality matching in a bipartite graph in time $O(n^{1.5}\sqrt{m/\log(n)})$," *Inf. Process. Lett.*, vol. 37, no. 4, pp. 237–240, 1991.

[26] K. McElvain, "LGSynth93 Benchmark Set: Version 4.0," online <http://www.cbl.ncsu.edu/pub/Benchmark_dirs/LGSynth93/doc/iwls93.ps>, May 1993.

[27] U. C. Group, "Espresso Examples," Online <ftp://ic.eecs.berkeley.edu/pub/Espresso/espresso-book-examples.tar.gz>, June 1993.